COMP 745 Semantics of Programming Languages

Course Notes

Peter Grogono

Original: December 1996
Revised: May 2002

Department of Computer Science
Concordia University

Contents

List of Figures

COMP 745 Semantics of Programming Languages

1 The Semantics of Binary Numerals

Semantics is "meaning". A computer can execute a program but does not (necessarily) know what the program "means". People understand programs intuitively: "I use this program to pay my bills." This is **not** the kind of semantics that we consider in this course.

We begin by studying examples of semantics. Later, we will address questions such as "What are semantics?", "Why do we need semantics?", and "What is the best kind of semantics?".

1.1 Grammar

Let us consider a very simple example: the "programming language" of binary numerals with addition. Here are some "programs" in this language:

$$\texttt{110} \qquad \texttt{010101} \qquad \texttt{101} \oplus \texttt{111} \tag{1}$$

We use the symbol "$\oplus$" to stand for addition of binary numerals. We expect the last of the three programs to be equivalent, in some sense, to the numeral `1100`, because $5 + 7 = 12$.

We call the language "BN" and we begin by defining its syntax. Since we are studying semantics, we are not interested in syntactic details, and we define the syntax in as simple a way as possible. For BN, we define the set $\mathbf{B}$ to be the smallest set that satisfies the recursive equation

$$B = \texttt{0} \mid \texttt{1} \mid B\texttt{0} \mid B\texttt{1} \mid B \oplus B$$

The three "programs" of (1) are examples of strings generated by this grammar. The empty string is **not** in the language. We do not use parentheses in the abstract syntax although parentheses are needed to distinguish $(x \oplus y) \oplus z$ and $x \oplus (y \oplus z)$.

We now consider three ways of assigning meaning to the strings of BN.

1.2 Denotational Semantics

We use binary numerals to represent natural numbers. For example, the numeral `101` represents the natural number 5. Formally, we say that "the denotation of `101` is 5". A **denotational semantics** is a system that provides a denotation in a mathematical domain for each string of a language.

We define the denotational semantics by means of one or more **semantic functions**. For BN, we need only one semantic function, $\mathcal{M} : \mathbf{B} \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers.

It is conventional in studies of semantics to enclose syntactic objects (in this example, members of $\mathbf{B}$) in "fat brackets" $[\![\cdot]\!]$. The formal way of writing "the denotation of `101` is 5" is $\mathcal{M}[\![\texttt{101}]\!] = 5$.

Figure 1 shows the definition of $\mathcal{M}$ by cases on the syntax; x and y stand for arbitrary members of $\mathbf{B}$.

The typographical distinctions in these equations are important: the `0` on the left is a binary numeral (member of B); the 0 on the right is a natural number (member of $\mathbb{N}$).

$$\begin{aligned}
\mathcal{M}[\![\,\texttt{0}\,]\!] &= 0 & (2)\\
\mathcal{M}[\![\,\texttt{1}\,]\!] &= 1 & (3)\\
\mathcal{M}[\![\,x\texttt{0}\,]\!] &= 2 \times \mathcal{M}[\![\,x\,]\!] & (4)\\
\mathcal{M}[\![\,x\texttt{1}\,]\!] &= 2 \times \mathcal{M}[\![\,x\,]\!] + 1 & (5)\\
\mathcal{M}[\![\,x \oplus y\,]\!] &= \mathcal{M}[\![\,x\,]\!] + \mathcal{M}[\![\,y\,]\!] & (6)
\end{aligned}$$

Figure 1: Semantic Equations for Binary Numerals

We can use the definition of $\mathcal{M}$ to convert any string of BN to a natural number. For example:

$$\begin{aligned}
\mathcal{M}[\![\,\texttt{101}\,]\!] &= 2 \times \mathcal{M}[\![\,\texttt{10}\,]\!] + 1 && \text{by (5)}\\
&= 2 \times (2 \times \mathcal{M}[\![\,\texttt{1}\,]\!]) + 1 && \text{by (4)}\\
&= 2 \times (2 \times 1) + 1 && \text{by (3)}\\
&= 5 && \text{Arithmetic}
\end{aligned}$$

Exercise 1 Use the definition of $\mathcal{M}$ to show that $\mathcal{M}[\![\,\texttt{101} \oplus \texttt{111}\,]\!] = \mathcal{M}[\![\,\texttt{1100}\,]\!]$. □

Exercise 2 Leading zeroes do not affect the value of a binary numeral. For example, `00101` denotes the same natural number (5) as `101`. Prove that, for any binary numeral x, $\mathcal{M}[\![\,\texttt{0}x\,]\!] = \mathcal{M}[\![\,x\,]\!]$. (Hint: use induction on the length of x.) □

1.3 Axiomatic Semantics

Another way of giving a meaning of binary numerals is to give a set of laws, or axioms, that binary numerals must satisfy. The result is an **axiomatic semantics**.

Note 1 (Equality) There are (at least) two possible interpretations of a formula such as $x = y$. We might be comparing the appearance of x and y, or we might be comparing their meanings. If we are comparing strings (*syntactic equality*), then the formula `101` = `000101` is false. If, on the other hand, we interpret the strings as binary numerals and compare their values (*semantic equality*), the formula is true.

Semantic equality is normal in mathematics. When we write "$2 + 2 = 4$", we do not mean that the string "$2 + 2$" is the same as the string "4". We implicitly assign meanings to the two strings and assume that the statement is about these meanings.

There are occasions when syntactic equality is useful. For example, an instructor marking a programming assignment might consider syntactic equality to be evidence of plagiarism, whereas semantic equality indicates that both students have written equivalent (and probably correct) programs. □

In the axioms shown in Figure 2, x, y, and z stand for members of $\mathbf{B}$. The symbol "=" denotes semantic equality.

$$\begin{aligned}
\mathtt{0} \oplus \mathtt{0} &= \mathtt{0} && (7)\\
\mathtt{0} \oplus \mathtt{1} &= \mathtt{1} && (8)\\
\mathtt{1} \oplus \mathtt{1} &= \mathtt{10} && (9)\\
\mathtt{0}x &= x && (10)\\
x \oplus y &= y \oplus x && (11)\\
x \oplus (y \oplus z) &= (x \oplus y) \oplus z && (12)\\
x\mathtt{0} \oplus y\mathtt{0} &= (x \oplus y)\,\mathtt{0} && (13)\\
x\mathtt{1} \oplus y\mathtt{0} &= (x \oplus y)\,\mathtt{1} && (14)\\
x\mathtt{1} \oplus y\mathtt{1} &= (x \oplus y \oplus \mathtt{1})\,\mathtt{0} && (15)
\end{aligned}$$

Figure 2: Axiomatic Semantics for Binary Numerals

In reading the axioms, note that they are about strings, not numbers. Consider, for example, how we can use the axioms to simplify a string containing an addition:

$$\begin{aligned}
\mathtt{11} \oplus \mathtt{10} &= (\mathtt{1} \oplus \mathtt{1})\,\mathtt{1} && \text{by (14)}\\
&= (\mathtt{10})\,\mathtt{1} && \text{by (9)}\\
&= \mathtt{101}
\end{aligned}$$

We can interpret this deduction as "$3 + 2 = 5$" but — note carefully! — the semantics does not say this: all it says is that the string "$\mathtt{11} \oplus \mathtt{10}$" is equivalent to the string "`101`".

Exercise 3 Show that the axioms of the Axiomatic Semantics are logical consequences of the Denotational Semantics. □

1.4 Operational Semantics

When we work with programming languages, we are interested in how programs are executed, or how the computer "operates". An **operational semantics** is a collection of rules that define a possible evaluation or execution of a program.

BN is a rather simple "programming language" but we can nevertheless give it a kind of operational semantics. The rules are similar to the axioms, but we replace "=" by "→" to say that evaluation has a direction. This gives a set of rules of the form $A \rightarrow B$, with the meaning "A can be rewritten as B": see Figure 3. We write "ϵ" to denote the empty string. A binary numeral is evaluated by using the rules to rewrite it until no rules apply.

We can use the operational semantics to evaluate $\mathtt{111} \oplus \mathtt{101}$ as follows (some steps are omitted):

$$\begin{aligned}
\mathtt{111} \oplus \mathtt{101} &\rightarrow ((\mathtt{11} \oplus \mathtt{10}) \oplus \mathtt{1})\mathtt{0}\\
&\rightarrow ((\mathtt{1} \oplus \mathtt{1})\mathtt{1} \oplus \mathtt{1})\mathtt{0}\\
&\rightarrow ((\mathtt{10})\mathtt{1} \oplus \mathtt{1})\mathtt{0}\\
&\rightarrow (((\mathtt{10} \oplus \epsilon) \oplus \mathtt{1})\mathtt{0})\mathtt{0}\\
&\rightarrow \mathtt{1100}
\end{aligned}$$

$$
\begin{aligned}
\epsilon \oplus x &\to x && (16)\\
x \oplus \epsilon &\to x && (17)\\
\mathtt{0}x &\to x \qquad (x \neq \epsilon) && (18)\\
x\mathtt{0} \oplus y\mathtt{0} &\to (x \oplus y)\,\mathtt{0} && (19)\\
x\mathtt{1} \oplus y\mathtt{0} &\to (x \oplus y)\,\mathtt{1} && (20)\\
x\mathtt{0} \oplus y\mathtt{1} &\to (x \oplus y)\,\mathtt{1} && (21)\\
x\mathtt{1} \oplus y\mathtt{1} &\to (x \oplus y \oplus \mathtt{1})\,\mathtt{0} && (22)
\end{aligned}
$$

Figure 3: Operational Semantics for Binary Numerals

Exercise 4 Give the derivation above in full, numbering each step with the number of the rule used. □

Exercise 5 Why is the empty string used in the operational semantics but not in the axiomatic semantics? □

Exercise 6 Why do we not obtain the operational semantics simply by changing "=" to "→" in the axiomatic semantics? □

Exercise 7 Show that the operational semantics is correct with respect to the denotational semantics. □

1.5 Comparison

We will use all three kinds of semantics — and yet other kinds — to examine simple programming languages. The simple example of BN, however, already tells us something about the usefulness of the different kinds of semantics.

- ▷ A denotational semantics tells us what program means, but does not (necessarily) tell us how to execute it.
- ▷ An axiomatic semantics describes properties that programs must have, but does not say what the program means or how to execute it.
- ▷ An operational semantics tells us how to execute a program, but does not tell us either the meaning of the program or any properties that it may possess.

A denotational semantics is more fundamental than an axiomatic semantics, because we can view the axioms as theorems that we can prove using the semantic function.

Reading Stoy (1977, pages 26–35) uses binary numerals to introduce semantic concepts.

2 The Semantics of a Simple Programming Language

In this section, we define a simple programming language and describe various ways of providing a meaning for programs in the language. The purpose of the section is to provide a bird's eye view of possible approaches; analysis of the difficulties that arise in defining semantics are postponed until later. We refer to the language as L_1.

2.1 Syntax

The set $\texttt{Num} = \texttt{0}, \texttt{1}, \texttt{2}, \ldots$ (typical members: m, n) consists of decimal numerals. The function $\mathrm{val} : \texttt{Num} \to \mathbb{N}$ maps numerals to the corresponding natural numbers.[1]

The set `Loc` (typical members: x, y) contains locations, or memory cells. Locations are represented in the program by variable names.

The set `Nexp` (typical member: e) contains numerical expressions. A numerical expression is a numeral, a variable, or the sum of two expressions:

$$e = n \mid x \mid e + e$$

The set `Bexp` (typical member: b) contains Boolean expressions:

$$b = \texttt{true} \mid \texttt{false} \mid x \mid e \le e$$

Note that the value of a variable may be either numeric or boolean.

We could easily add other operators, such as those of Figure 15, to the language, but this would make all the definitions longer without adding any new features of interest.

The set `Com` (typical member: c) contains commands.

$$c = \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ b\ \texttt{do}\ c$$

Note 2 It is conventional in semantic studies to refer to constructs in programming languages that have side-effects as "commands" rather than "statements", to avoid confusion with the logical concept of "statement". □

We will use the following simple program as a running example:

$$x := 0; \texttt{while}\ x \le 1\ \texttt{do}\ x := x + 1 \tag{23}$$

2.2 Informal Semantics

An informal semantics describes the meaning of each construct of the programming language using a natural language, such as English. For example, L_1 has a `while` statement that is similar to the `while` statement of Pascal. The Pascal Standard (which gives an informal definition of Pascal) has this to say about `while` statements:

[1] Functions such as val are needed for precision but are somewhat pedantic for semantic studies. Later on, we will not bother with such "obvious" functions.

The while-statement

```
while b do
   body
```

shall be equivalent to

```
begin
   if b then
     repeat
       body
     until not (b)
end
```

The Standard explains the `repeat` statement like this:

> The statement-sequence of the repeat-sequence shall be repeatedly executed (except as modified by the execution of a goto-statement) until the Boolean-expression of the repeat-statement yields the value true on completion of the statement-sequence. The statement-sequence shall be executed at least once, because the Boolean-expression is evaluated after execution of the statement-sequence.

This kind of explanation does not "explain" very much to someone who is not already familiar with the concepts.

2.3 Operational Semantics

An operational semantics explains how to translate programs into a simpler language, usually defined for an abstract machine. Figure 5 gives a set of rules for translating L_1 programs into an abstract stack machine. Figure 4 defines the instructions of the abstract stack machine (informally!).

`push` N	Push integer N onto the stack.
`push` $M[x]$	Push memory location x onto the stack.
`pop` $M[x]$	Pop the stack, storing the top element at memory location x.
`add`	Pop the top two locations of the stack, add them, and push the result onto the stack.
`le`	Pop the top two locations of the stack, compare them, and push the result (0 or 1) onto the stack.
`j` α	Jump to label α.
`jz` α	Pop the stack; if the result is zero, jump to label α.
`lab` α	The label α.
`hlt`	Terminate execution.

Figure 4: Abstract Stack Machine Instructions

The notation $\mathcal{C} [\![p]\!] \to L$ means that the program fragment p is compiled to a list of instructions L. Angle brackets convert an instruction to a list with one component: for example, $\langle$`add`$\rangle$ is a list containing the instruction `add`. The operator ";" is used to concatenate lists.

Note that the expression x compiles to $\langle \texttt{push}\ M[x] \rangle$. This is a simplification: we should provide a mapping from program variables to memory addresses, and use the addresses in the generated code. Also, the compiler must generate unique labels, not just the labels α and β.

$$\begin{array}{rcl}
\mathcal{C}[\![n]\!] & \to & \langle \texttt{push}\ \mathrm{val}\, n \rangle \\
\mathcal{C}[\![\texttt{true}]\!] & \to & \langle \texttt{push}\ 1 \rangle \\
\mathcal{C}[\![\texttt{false}]\!] & \to & \langle \texttt{push}\ 0 \rangle \\
\mathcal{C}[\![x]\!] & \to & \langle \texttt{push}\ M[x] \rangle \\
\mathcal{C}[\![e_1 + e_2]\!] & \to & \mathcal{C}[\![e_1]\!] ; \mathcal{C}[\![e_2]\!] ; \langle \texttt{add} \rangle \\
\mathcal{C}[\![e_1 \le e_2]\!] & \to & \mathcal{C}[\![e_1]\!] ; \mathcal{C}[\![e_2]\!] ; \langle \texttt{le} \rangle \\
\mathcal{C}[\![\texttt{skip}]\!] & \to & \langle \rangle \\
\mathcal{C}[\![x := e]\!] & \to & \mathcal{C}[\![e]\!] ; \langle \texttt{pop}\ M[x] \rangle \\
\mathcal{C}[\![c_1 ; c_2]\!] & \to & \mathcal{C}[\![c_1]\!] ; \mathcal{C}[\![c_2]\!] \\
\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] & \to & \mathcal{C}[\![b]\!] ; \langle \texttt{jz}\ \alpha \rangle ; \mathcal{C}[\![c_1]\!] ; \langle \texttt{j}\ \beta \rangle ; \langle \texttt{lab}\ \alpha \rangle ; \mathcal{C}[\![c_2]\!] ; \langle \texttt{lab}\ \beta \rangle \\
\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!] & \to & \langle \texttt{lab}\ \alpha \rangle ; \mathcal{C}[\![b]\!] ; \langle \texttt{jz}\ \beta \rangle ; \mathcal{C}[\![c]\!] ; \langle \texttt{j}\ \alpha \rangle ; \langle \texttt{lab}\ \beta \rangle
\end{array}$$

Figure 5: Operational Semantics for L_1

$$\begin{array}{l}
\texttt{push}\ 0; \texttt{pop}\ M[x]; \texttt{lab}\ 1; \texttt{push}\ M[x]; \texttt{push}\ 1; \texttt{le}; \texttt{jz}\ 2; \\
\texttt{push}\ M[x]; \texttt{push}\ 1; \texttt{add}; \texttt{pop}\ M[x]; \texttt{j}\ 1; \texttt{lab}\ 2; \texttt{hlt}
\end{array}$$

Figure 6: Operational Semantic Value of the Example Program

Figure 6 shows the result of compiling the example program (23). Generation of the final instruction, `hlt`, is not shown in Figure 5.

Exercise 8 Show, by induction on the operational semantics of L_1 (Figure 5), that the stack code of any program contains only properly nested sequences of `push` and `pop` instructions. In other words, if we omit instructions other than `push` and `pop`, the program has the form PP^* where

$$PP \quad = \quad \langle \rangle \mid \texttt{push}\ x;\ PP;\ \texttt{pop}\ M[y].$$

□

2.4 States

The remaining kinds of semantics in this section require the concept of *state*, reflecting that the effect of executing a program is to change the state of a hypothetical machine. The state is modelled as a partial function from variable names (locations) to the contents of the corresponding location. Since a location can hold either a number or a boolean value, a state is a function with the type $\texttt{Loc} \to \mathbb{N} \cup \mathbb{T}$. ($\mathbb{N} = \{ 0, 1, 2, \dots \}$ is the set of *natural numbers* and

$\mathbb{T} = \{ \mathsf{T}, \mathsf{F} \}$ is the set of *truth values*.) We write σ to denote a typical state and Σ to denote the set of all states.

Our notation for states is as follows. If we know nothing at all about the state, we write $\langle\rangle$. For a state in which we know that $x = 3$ and $y = \mathsf{T}$, we write $\langle x = 3, y = \mathsf{T}\rangle$.

We express changes to the state using the function override notation of **Z**. The equation $\sigma' = \sigma \oplus \langle x = 1\rangle$ means that σ' is a state which is the same as σ except that x has the value 1.

Since a state is a function, we can apply it to an argument. For example, $\langle x = 2, y = 5\rangle(x) = 2$. Applying a state to a variable that is not defined in the state is an error. For example, $\langle x = 2, y = 5\rangle(z)$ is an error.[2]

We will assume that the evaluation of an expression does not change the state.

2.5 Transition Semantics

A transition semantics describes the effect of each statement of the language as a state transition. A semantic rule has the form

$$(p, \sigma) \rightarrow (p', \sigma')$$

and is interpreted as a transition from a situation in which program p must be executed in state σ to a situation in which program p' must be executed in state σ'. If the execution of p completes the program, we write $(p, \sigma) \rightarrow \sigma'$, indicating that σ' is the final state yielded by the program.

We do not show the details of expression evaluation in the transitional semantics shown in Figure 7. The form $\mathcal{V}[\![e]\!]\sigma \rightarrow E$ means that evaluation of the expression e in state σ yields the value E.

Figure 8 provides a simplified trace of the execution of the example program (23) using the transition semantics.

Exercise 9 Explain why there are two rules for composition ((26) and (27)) rather than just one. $\Box$

2.6 Denotational Semantics

A *denotational semantics* (also sometimes know as *mathematical semantics* or *classical semantics*) maps programs to elements of a mathematical domain. For example, in Section 1.2, we expressed the semantics of binary numerals as a mapping from binary numerals to natural numbers.

Clearly, natural numbers are not a suitable domain for programs. Instead, we use functions from states to states as the domain. That is, each expression of L_1 has a value in $\mathbb{N} \cup \mathbb{T}$ and each statement has a value in $\Sigma \rightarrow \Sigma$. The functions we will need are

$$\begin{aligned} \mathcal{V}[\![\cdot]\!] &: \mathtt{Nexp} \cup \mathtt{Bexp} \rightarrow (\Sigma \rightarrow \mathbb{N} \cup \mathbb{T}) \\ \mathcal{C}[\![\cdot]\!] &: \mathtt{Com} \rightarrow (\Sigma \rightarrow \Sigma) \end{aligned}$$

[2]Errors arise because states are *partial* functions. We will discuss what to do with them later.

$$(\texttt{skip}, \sigma) \to \sigma \tag{24}$$

$$\frac{\mathcal{V}[\![e]\!]\sigma \to E}{(x := e, \sigma) \to \sigma \oplus \langle x = E \rangle} \tag{25}$$

$$\frac{(c_1, \sigma) \to (c_1', \sigma')}{(c_1; c_2, \sigma) \to (c_1'; c_2, \sigma')} \tag{26}$$

$$\frac{(c_1, \sigma) \to \sigma'}{(c_1; c_2, \sigma) \to (c_2, \sigma')} \tag{27}$$

$$\frac{\mathcal{V}[\![b]\!]\sigma \to \mathsf{T}}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma) \to (c_1, \sigma)} \tag{28}$$

$$\frac{\mathcal{V}[\![b]\!]\sigma \to \mathsf{F}}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma) \to (c_2, \sigma)} \tag{29}$$

$$\frac{\mathcal{V}[\![b]\!]\sigma \to \mathsf{T}}{(\texttt{while}\ b\ \texttt{do}\ c, \sigma) \to (c; \texttt{while}\ b\ \texttt{do}\ c, \sigma)} \tag{30}$$

$$\frac{\mathcal{V}[\![b]\!]\sigma \to \mathsf{F}}{(\texttt{while}\ b\ \texttt{do}\ c, \sigma) \to \sigma} \tag{31}$$

Figure 7: Transition Semantics for L_1

$$\begin{array}{ll}
 & x := 0; \texttt{while}\ x \leq 1\ \texttt{do}\ x := x + 1, \langle\rangle \\
\to & \texttt{while}\ x \leq 1\ \texttt{do}\ x := x + 1, \langle x = 0 \rangle \\
\to & x := x + 1; \texttt{while}\ x \leq 1\ \texttt{do}\ x := x + 1, \langle x = 0 \rangle \\
\to & \texttt{while}\ x \leq 1\ \texttt{do}\ x := x + 1, \langle x = 1 \rangle \\
\to & x := x + 1; \texttt{while}\ x \leq 1\ \texttt{do}\ x := x + 1, \langle x = 1 \rangle \\
\to & \texttt{while}\ x \leq 1\ \texttt{do}\ x := x + 1, \langle x = 2 \rangle \\
\to & \langle x = 2 \rangle
\end{array}$$

Figure 8: Transition semantics applied to the example program

Strictly, we should write the semantic equations as $\mathcal{V}[\![\cdot]\!] = \lambda\sigma \,.\, v$, because the value of $\mathcal{V}[\![\cdot]\!]$ is a function. In this section, we will use the less formal notation $\mathcal{V}[\![\cdot]\!]\,\sigma = v$.

Figure 9 shows a denotational semantics for L_1. In (41) and (42), the notation $x \to y,\ z$ means "if x yields T, then y, else z".

$$
\begin{aligned}
\mathcal{V}[\![n]\!]\,\sigma &= \mathrm{val}\,n && (32)\\
\mathcal{V}[\![\mathtt{true}]\!]\sigma &= \mathsf{T} && (33)\\
\mathcal{V}[\![\mathtt{false}]\!]\sigma &= \mathsf{F} && (34)\\
\mathcal{V}[\![x]\!]\,\sigma &= \sigma(x) && (35)\\
\mathcal{V}[\![e_1 + e_2]\!]\,\sigma &= \mathcal{V}[\![e_1]\!]\,\sigma + \mathcal{V}[\![e_2]\!]\,\sigma && (36)\\
\mathcal{V}[\![e_1 \le e_2]\!]\,\sigma &= \mathcal{V}[\![e_1]\!]\,\sigma \le \mathcal{V}[\![e_2]\!]\,\sigma && (37)\\
\mathcal{C}[\![\mathtt{skip}]\!]\sigma &= \sigma && (38)\\
\mathcal{C}[\![x := E]\!]\sigma &= \sigma \oplus \langle x = E\rangle && (39)\\
\mathcal{C}[\![c_1; c_2]\!]\,\sigma &= \mathcal{C}[\![c_2]\!]\,(\mathcal{C}[\![c_1]\!]\,\sigma) && (40)\\
\mathcal{C}[\![\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2]\!]\,\sigma &= \mathcal{V}[\![b]\!]\,\sigma \to \mathcal{C}[\![c_1]\!]\sigma,\ \ \mathcal{C}[\![c_2]\!]\,\sigma && (41)\\
\mathcal{C}[\![\mathtt{while}\ b\ \mathtt{do}\ c]\!]\sigma &= \mathcal{V}[\![b]\!]\sigma \ \to \mathcal{C}[\![c; \mathtt{while}\ b\ \mathtt{do}\ c]\!]\,\sigma,\ \sigma && (42)
\end{aligned}
$$

Figure 9: Denotational semantics for L_1

Notice that, in some cases, we can easily "factor out" the state. For example, we can write (40) in the form

$$\mathcal{C}[\![c_1; c_2]\!] = \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!]$$

using $\circ$ to denote functional composition.

Unfortunately, the semantics given in Figure 9 is not a denotational semantics in the strict sense because it is not compositional.

Definition 3 (Compositional) *A semantics is* **compositional** *if it assigns a meaning to every structured syntactic item in terms of the meanings of the components of the item.*

For example, (40) is compositional because it defines the meaning of the compound command $(c_1; c_2)$ in terms of the meanings of c_1 and c_2. Similarly, (41) is compositional because it defines the meaning of `if` b `then` c_1 `else` c_2 in terms of the meanings of b, c_1, and c_2. But (42) is not compositional because `while` b `do` c appears on both sides of the defining equation.

In fact, (42) is a circular definition for some programs. Using the fact that `skip` is the identity function, we can show that

$$
\begin{aligned}
\mathcal{C}[\![\mathtt{while\ true\ do\ skip}]\!]\sigma &= \mathcal{V}[\![\mathtt{true}]\!]\sigma \ \to \mathcal{C}[\![\mathtt{skip}\ ; \mathtt{while\ true\ do\ skip}]\!]\sigma,\ \sigma\\
&= \mathcal{C}[\![\mathtt{skip}; \mathtt{while\ true\ do\ skip}]\!]\sigma\\
&= \mathcal{C}[\![\mathtt{while\ true\ do\ skip}]\!]\sigma
\end{aligned}
$$

We can obtain a compositional definition for `while` statements by using a trick. Consider the program X, recursively defined by

$$X = \mathtt{if}\ b\ \mathtt{then}\ (c; X)\ \mathtt{else}\ \mathtt{skip}.$$

Intuitively, X behaves as we would expect `while` b `do` c to behave. In fact, we can use it, in slightly modified form, to define the intended interpretation of `while` b `do` c:

$$\begin{aligned} \mathcal{C}\,[\![\, \texttt{while}\; b\; \texttt{do}\; c \,]\!]\,\sigma &= X\sigma \\ &\quad \text{where} \quad X\sigma = \mathcal{C}\,[\![\, b \,]\!]\sigma \;\rightarrow (X \circ \mathcal{C}\,[\![\, c \,]\!])\sigma,\; \sigma \end{aligned} \tag{43}$$

This definition, however, introduces a new problem: recursion. In order to justify definition (43), we will have to show that recursive definitions are well-formed: in this case, we must be confident that X has a unique, meaningful value.

The denotational semantics is incomplete in other ways. For example, it provides a meaning for the program $[\![\, x := 1; y := 1 \,]\!]$ but not for the program $[\![\, y := x \,]\!]$. It is not enough to say that variables must be initialized before they are used, because we want a semantics that assigns a meaning to every program that is *syntactically* correct.

2.7 Natural Semantics

A natural semantics is similar to a transition semantics. Each rule, however, expresses the *complete* evaluation of an expression or execution of a command, rather than a single step. The rules are presented as an inference system, with axioms and inferences.

Figure 10 shows the natural semantics of L_1 expressed as an inference system. Formulas of the inference system have the form

$$f\,[\![\, p \,]\!]\; \sigma \rightarrow r$$

where f is a semantic function and r is the result of evaluating the program p in state σ. The semantic functions we need for L_1 are as follows:

$\mathcal{V}\,[\![\, \cdot \,]\!]$	$: (\texttt{Nexp} \cup \texttt{Bexp}) \times \Sigma$	$\rightarrow$	$(\mathbb{N} \cup \mathbb{T})$	Numeric expressions
$\mathcal{C}\,[\![\, \cdot \,]\!]$	$: \texttt{Com} \times \Sigma$	$\rightarrow$	Σ	Commands

The semantic value of an expression is a natural number in $\mathbb{N}$ or a truth-value in $\mathbb{T}$. The semantic value of a command is a new state.

Figure 11 shows a derivation using the operational semantics. The derivation shows that execution of the example program (23) in any state terminates in a state in which $x = 2$.

The proof in Figure 11 is tree-structured. The conclusion is the line labelled 1. It is a consequence of the lines labelled 1.1 and 1.2, corresponding to an application of rule (52). Similarly, line 1.2 is a consequence of lines 1.2.1, 1.2.2, and 1.2.3, corresponding to an application of rule (55).

It is possible for two programs to have the same effect on the state. This suggests an equivalence relation, $\sim$, defined as follows:

$$c_1 \sim c_2 \quad \Longleftrightarrow \quad \forall \sigma, \sigma' \,.\, (\mathcal{C}\,[\![\, c_1 \,]\!]\, \sigma \rightarrow \sigma' \iff \mathcal{C}\,[\![\, c_2 \,]\!]\, \sigma \rightarrow \sigma')$$

In words: c_1 and c_2 are equivalent if they have the same effect on all input states. For example, $x := 1; y := 2$ and $y := 2; x := 1$ are equivalent programs.

Exercise 10 Prove that, for any programs c_1, c_2, and c_3:

$$(c_1; c_2); c_3 \sim c_1; (c_2; c_3).$$

$$\mathcal{V}[\![\, \mathtt{n} \,]\!]\ \sigma \rightarrow \mathrm{val}\, n \tag{44}$$

$$\mathcal{V}[\![\, \mathtt{true} \,]\!]\ \sigma \rightarrow \mathsf{T} \tag{45}$$

$$\mathcal{V}[\![\, \mathtt{false} \,]\!]\ \sigma \rightarrow \mathsf{F} \tag{46}$$

$$\mathcal{V}[\![\, x \,]\!]\ \sigma \rightarrow \sigma(x) \tag{47}$$

$$\mathcal{C}[\![\, \mathtt{skip} \,]\!]\ \sigma \rightarrow \sigma \tag{48}$$

$$\frac{\mathcal{V}[\![\, e_1 \,]\!]\ \sigma \rightarrow n_1 \qquad \mathcal{V}[\![\, e_2 \,]\!]\ \sigma \rightarrow n_2}{\mathcal{V}[\![\, e_1 + e_2 \,]\!]\ \sigma \rightarrow n_1 + n_2} \tag{49}$$

$$\frac{\mathcal{V}[\![\, e_1 \,]\!]\ \sigma \rightarrow n_1 \qquad \mathcal{V}[\![\, e_2 \,]\!]\ \sigma \rightarrow n_2}{\mathcal{V}[\![\, e_1 \leq e_2 \,]\!]\ \sigma \rightarrow n_1 \leq n_2} \tag{50}$$

$$\frac{\mathcal{V}[\![\, e \,]\!]\ \sigma \rightarrow E}{\mathcal{C}[\![\, x := e \,]\!]\ \sigma \rightarrow \sigma \oplus \langle x = E \rangle} \tag{51}$$

$$\frac{\mathcal{C}[\![\, c_1 \,]\!]\ \sigma_1 \rightarrow \sigma_2 \qquad \mathcal{C}[\![\, c_2 \,]\!]\ \sigma_2 \rightarrow \sigma_3}{\mathcal{C}[\![\, c_1; c_2 \,]\!]\ \sigma_1 \rightarrow \sigma_3} \tag{52}$$

$$\frac{\mathcal{V}[\![\, b \,]\!]\ \sigma \rightarrow \mathsf{T}}{\mathcal{C}[\![\, \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \,]\!]\ \sigma \rightarrow \mathcal{C}[\![\, c_1 \,]\!]\ \sigma} \tag{53}$$

$$\frac{\mathcal{V}[\![\, b \,]\!]\ \sigma \rightarrow \mathsf{F}}{\mathcal{C}[\![\, \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \,]\!]\ \sigma \rightarrow \mathcal{C}[\![\, c_2 \,]\!]\ \sigma} \tag{54}$$

$$\frac{\mathcal{V}[\![\, b \,]\!]\ \sigma_0 \rightarrow \mathsf{T} \qquad \mathcal{C}[\![\, c \,]\!]\ \sigma_0 \rightarrow \sigma_1 \qquad \mathcal{C}[\![\, \mathtt{while}\ b\ \mathtt{do}\ c \,]\!]\ \sigma_1 \rightarrow \sigma_2}{\mathcal{C}[\![\, \mathtt{while}\ b\ \mathtt{do}\ c \,]\!]\ \sigma_0 \rightarrow \sigma_2} \tag{55}$$

$$\frac{\mathcal{V}[\![\, b \,]\!]\ \sigma \rightarrow \mathsf{F}}{\mathcal{C}[\![\, \mathtt{while}\ b\ \mathtt{do}\ c \,]\!]\ \sigma \rightarrow \sigma} \tag{56}$$

Figure 10: Natural Semantics for L_1

1.1.1	$\mathcal{V}\,[\![\,0\,]\!]\ \langle\rangle \to 0$
1.1	$\mathcal{C}\,[\![\,x := 0\,]\!]\ \langle\rangle \to \langle x = 0\rangle$
1.2.1.1	$\mathcal{V}\,[\![\,x\,]\!]\ \langle x = 0\rangle \to 0$
1.2.1.2	$\mathcal{V}\,[\![\,1\,]\!]\ \langle x = 0\rangle \to 1$
1.2.1	$\mathcal{V}\,[\![\,x \le 1\,]\!]\ \langle x = 0\rangle \to \mathsf{T}$
1.2.2.1.1	$\mathcal{V}\,[\![\,x\,]\!]\ \langle x = 0\rangle \to 0$
1.2.2.1.2	$\mathcal{V}\,[\![\,1\,]\!]\ \langle x = 0\rangle \to 1$
1.2.2.1	$\mathcal{V}\,[\![\,x + 1\,]\!]\ \langle x = 0\rangle \to 1$
1.2.2	$\mathcal{C}\,[\![\,x := x + 1\,]\!]\ \langle x = 0\rangle \to \langle x = 1\rangle$
1.2.3.1.1	$\mathcal{V}\,[\![\,x\,]\!]\ \langle x = 1\rangle \to 1$
1.2.3.1.2	$\mathcal{V}\,[\![\,1\,]\!]\ \langle x = 1\rangle \to 1$
1.2.3.1	$\mathcal{V}\,[\![\,x \le 1\,]\!]\ \langle x = 1\rangle \to \mathsf{T}$
1.2.3.2.1.1	$\mathcal{V}\,[\![\,x\,]\!]\ \langle x = 1\rangle \to 1$
1.2.3.2.1.2	$\mathcal{V}\,[\![\,1\,]\!]\ \langle x = 1\rangle \to 1$
1.2.3.2.1	$\mathcal{V}\,[\![\,x + 1\,]\!]\ \langle x = 1\rangle \to 2$
1.2.3.2	$\mathcal{C}\,[\![\,x := x + 1\,]\!]\ \langle x = 1\rangle \to \langle x = 2\rangle$
1.2.3.3.1.1	$\mathcal{V}\,[\![\,x\,]\!]\ \langle x = 2\rangle \to 2$
1.2.3.3.1.2	$\mathcal{V}\,[\![\,1\,]\!]\ \langle x = 2\rangle \to 1$
1.2.3.3.1	$\mathcal{V}\,[\![\,x \le 1\,]\!]\ \langle x = 2\rangle \to \mathsf{F}$
1.2.3.3	$\mathcal{C}\,[\![\,\texttt{while}\ x \le 1\ \texttt{do}\ x := x + 1\,]\!]\ \langle x = 2\rangle \to \langle x = 2\rangle$
1.2.3	$\mathcal{C}\,[\![\,\texttt{while}\ x \le 1\ \texttt{do}\ x := x + 1\,]\!]\ \langle x = 1\rangle \to \langle x = 2\rangle$
1.2	$\mathcal{C}\,[\![\,\texttt{while}\ x \le 1\ \texttt{do}\ x := x + 1\,]\!]\ \langle x = 0\rangle \to \langle x = 2\rangle$
1	$\mathcal{C}\,[\![\,x := 0; \texttt{while}\ x \le 1\ \texttt{do}\ x := x + 1\,]\!]\ \langle\rangle \to \langle x = 2\rangle$

Figure 11: Natural semantics applied to the example program

□

Exercise 11 What is wrong with the following definition of equivalence in the natural semantics of L_1? Would this definition be suitable for the denotational semantics?

$$c_1 \sim c_2 \iff \forall\sigma\,.\,(\mathcal{C}\,[\![\,c_1\,]\!]\,\sigma = \mathcal{C}\,[\![\,c_2\,]\!]\,\sigma).$$

□

Exercise 12 Use an argument based on derivation structure to show that there can be no state σ' such that $\mathcal{C}\,[\![\,\texttt{while true do skip}\,]\!]\,\sigma \to \sigma'$. □

2.8 Axiomatic Semantics

We express the meaning of statements of L_1 axiomatically using statements of the form

$$\{\,P\,\}\ S\ \{\,Q\,\} \tag{57}$$

The braces $\{\cdot\}$ contain comments, as in Pascal, and S is a statement. The comments contain **state predicates**: a state predicate $P(\sigma)$ is either true or false for every $\sigma \in \Sigma$. Informally, (57) means:

> If $P(\sigma)$ is true, and S is executed in state σ then, when S terminates yielding state σ', $Q(\sigma')$ is true.

We call P the *precondition* and Q the *postcondition*.

Formally, the axiomatic semantics is a first-order theory consisting of predicate calculus extended with formulas of the form $\{\,P\,\}\ S\ \{\,Q\,\}$. Floyd (1967) first coupled assertions to programs by annotating flowcharts; Hoare (1969) introduced the formal theory. We could write the formulas of the axiomatic smentics as $P\ [\![S\ ]\!]\ Q$, which would be closer to Hoare's original notation and would also follow our convention of using $[\![\cdot]\!]$ for program text, but the notation given has become conventional.

Note 4 Informally, a predicate P is *stronger* than another predicate Q if $P \Rightarrow Q$. The strongest predicate is `false`, because `false` $\Rightarrow P$ for all predicates P. The weakest predicate is `true`, because $P \Rightarrow$ `true` for all predicates P. Note that $P \vee Q$ is weaker than P because $\models P \Rightarrow P \vee Q$ and that $P \wedge Q$ is stronger than P because $\models P \wedge Q \Rightarrow P$.

In particular, `true` is the weakest precondition ($\{$ `true` $\}\ S\ \{\,Q\,\}$ says that S executed in any state ensures Q) and the weakest post-condition ($\{\,P\,\}\ S\ \{$ `true` $\}$ says that S executed in state P can do anything at all). □

$$\{\,P\,\}\ \texttt{skip}\ \{\,P\,\} \tag{58}$$

$$\{\,P[E/x]\,\}\ x := E\ \{\,P\,\} \tag{59}$$

$$\frac{\{\,P\,\}\ S_1\ \{\,R\,\} \qquad \{\,R\,\}\ S_2\ \{\,Q\,\}}{\{\,P\,\}\ S_1; S_2\ \{\,Q\,\}} \tag{60}$$

$$\frac{\{\,P \wedge B\,\}\ S_1\ \{\,Q\,\} \qquad \{\,P \wedge \neg B\,\}\ S_2\ \{\,Q\,\}}{\{\,P\,\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\,Q\,\}} \tag{61}$$

$$\frac{P \Rightarrow I \qquad \{\,I \wedge B\,\}\ S\ \{\,I\,\} \qquad I \wedge \neg B \Rightarrow Q}{\{\,P\,\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{\,Q\,\}} \tag{62}$$

$$\frac{\{\,P\,\}\ S\ \{\,Q\,\} \qquad P' \Rightarrow P}{\{\,P'\,\}\ S\ \{\,Q\,\}} \tag{63}$$

$$\frac{\{\,P\,\}\ S\ \{\,Q\,\} \qquad Q \Rightarrow Q'}{\{\,P\,\}\ S\ \{\,Q'\,\}} \tag{64}$$

Figure 12: Axiomatic semantics for L_1

Figure 12 shows the axiomatic semantics of L_1 statements in the form of axioms and inference rules. We refer to (63) as "strengthening the precondition" and to (64) as "weakening the postcondition". In (59), the expression $P[E/x]$ means "P with all free occurrences of x replaced by E."

The assignment rule, (59), looks back-to-front. It is, in fact, often used from right to left: from a given postcondition, we use it to determined the required precondition. For example, from

$$\{\,P\,\}\ x := x+1\ \{\,x \geq 3\,\}$$

we infer

$$\begin{aligned} P &\equiv (x \geq 3)[x+1/x] \\ &\equiv x+1 \geq 3 \\ &\equiv x \geq 2 \end{aligned}$$

Consider next the statement `while` $y \geq 0$ `do` $x := x+1$ in a state in which $x = 0$ and $y = 0$. Let

$$\begin{aligned} P &\equiv x = 0 \wedge y = 0 \\ I &\equiv x \geq 0 \\ B &\equiv y \geq 0 \end{aligned}$$

Then we have

$$\begin{gathered} P \Rightarrow I \\ \{\,x \geq 0 \wedge y \geq 0\,\}\ x := x+1\ \{\,x \geq 0\,\} \\ I \wedge \neg B \equiv x \geq 0 \wedge y < 0 \equiv Q \end{gathered}$$

and so, by (62),

$$\{\,x = 0 \wedge y = 0\,\}\ \texttt{while}\ y \geq 0\ \texttt{do}\ x := x+1\ \{\,x \geq 0 \wedge y < 0\,\}.$$

This is rather surprising: we have proved that a loop which increments x can give y a negative value! In fact, of course, the loop does not terminate.

Axiomatic semantics provides *partial correctness*: it tells us about the final state only if the statement terminates. This contrasts with *total correctness*, which tells us both that the statement terminates and what the final state is.

Exercise 13 Deduce from the axiomatic semantics that the assignment $x := E$ cannot change the value of any variable other than x. $\square$

2.9 Weakest Precondition Semantics

Weakest precondition semantics is similar in some ways to axiomatic semantics. It was introduced by Dijkstra (1976) and is described (with changes of notation) by Meyer (1990).

Consider how we might use the formula $\{\,P\,\}\ S\ \{\,Q\,\}$ in program development. The *goal* of the program is to establish the postcondition Q. For example, the postcondition of a square root program might be $Q \equiv x \approx \sqrt{2}$. The obvious question is: what is the *weakest precondition* needed for the execution of a command S to ensure Q?

The question suggests a function, $\mathcal{W}$, that, given a command S and a postcondition Q yields a condition $\mathcal{W}(S, Q)$ with the required property. Conditions have type $\Sigma \to \mathbb{T}$ and so $\mathcal{W}$ has type `Com` $\to (\Sigma \to \mathbb{T}) \to (\Sigma \to \mathbb{T})$.

Figure 13 shows laws for the weakest precondition function and Figure 14 gives rules for evaluating it.

$$\begin{array}{rcll}
\mathcal{W}(S, \mathsf{F}) & = & \mathsf{F} & (65) \\
Q \Rightarrow Q' & \vdash & \mathcal{W}(S, Q) \Rightarrow W(S, Q') & (66) \\
\mathcal{W}(S, Q \wedge Q') & \Longleftrightarrow & \mathcal{W}(S, Q) \wedge \mathcal{W}(S, Q') & (67) \\
\mathcal{W}(S, Q \vee Q') & \Longleftrightarrow & \mathcal{W}(S, Q) \vee \mathcal{W}(S, Q') & (68)
\end{array}$$

Figure 13: Laws for the weakest precondition function

$$\begin{array}{rcll}
\mathcal{W}(\texttt{skip}, Q) & = & Q & (69) \\
\mathcal{W}(x := E, Q) & = & Q[E/x] & (70) \\
\mathcal{W}(c_1; c_2, Q) & = & \mathcal{W}(c_1, \mathcal{W}(c_2, Q)) & (71) \\
\mathcal{W}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, Q) & = & \mathcal{W}(c_1, Q) \wedge b \vee \mathcal{W}(c_2, Q) \wedge \neg b & (72) \\
\mathcal{W}(\texttt{while}\ b\ \texttt{do}\ c, Q) & = & \exists n \,.\, G(n) & (73) \\
 & & \begin{array}{lrcl} \text{where} & G(0) & = & \neg b \wedge Q \\ \text{and} & G(i) & = & b \wedge \mathcal{W}(c, G(i-1)) \end{array} & (74)
\end{array}$$

Figure 14: Evaluating the weakest precondition function

▷ Dijkstra calls (65) the *Law of the Excluded Miracle*: informally, it says that there are no conditions under which an arbitrary program could yield the postcondition `false`. It is tempting to assume that $\mathcal{W}(S, \mathsf{T})$ would be `true`, but this is not the case: it is the weakest condition under which S will terminate.

▷ To see how the assignment law, (70), works, consider: $\mathcal{W}(x = 2n + 3, x \geq 5) = n \geq 1$.

▷ The law for loops, (73), is not very helpful in the form given here. For practical applications, we use an alternative rule that is based on invariants and is similar to the inference rule for `while` commands in the axiomatic semantics, (62).

▷ The law of disjunction, (68), does not hold for non-deterministic programs (see Exercise 14).

Exercise 14 Suppose that $S \equiv \texttt{Toss}\ x$ is a program that simulates the tossing of a coin, setting x to either *heads* or *tails* non-deterministically. Let $Q \equiv (x = \textit{heads})$ and $Q' \equiv (x = \textit{tails})$. Show that (68) does not hold. □

2.10 Discussion

The *operational semantics* provides, for each L_1 program, an equivalent program in another language — L_2, say. We have not formalized L_1 completely until we have provided a formal semantics for L_2. Operational semantics are sometimes justified by the claim that that L_2 is so simple and "intuitive" that we do not need to formalize it. But this defence fails on two grounds. First, we cannot write rigorous proofs without an adequate theory; second, the "simplicity" of L_2 is illusory: consider, for example, discussing the semantics of recursive data structures using a simple stack language.

The *transition semantics* is an improvement over the operational semantics because it abstracts away from programs as sequences of instructions. Instead, the meaning of a program is expressed by describing the effect of executing the program on the state. The transition semantics is of little use to a programmer using L_1 but it might be of some use to a programmer writing a compiler for L_1.

The *denotational semantics* takes the idea of state transitions one step further: programs that affect states can be considered as functions on the state. If we write the semantic equations in a formal way, so that, for example, (35) becomes

$$\mathcal{V}\,[\![x]\!] = \lambda\sigma\,.\,\sigma(x)$$

we have actually achieved our stated intention of providing a meaning to each program in the form of a mathematical object. The denotational semantics might be useful to both designers and implementors of L_1.

The *natural semantics* is little more than an alternative presentation of the denotational semantics. It is sometimes easier to reason with a natural semantics than with a denotational semantics, because the inference rules suggest arguments based on proof structure.

The *axiomatic semantics* is intended to help programmers develop programs. The axioms and inference rules provide as much information as the programmer needs to write programs, but no more. The axiomatic semantics are not fundamental, because they can be derived from the denotational semantics.

The *weakest precondition semantics* extends the idea of the axiomatic semantics by providing a means of calculating the precondition from a given postcondition. In principle, programming becomes an semi-automatic, goal-directed activity: starting from a desired postcondition Q, we discover a program S such that $\mathcal{W}(S, Q) = \mathsf{T}$. In practice, of course, programming experience is essential, and weakest precondition calculations merely provide precise confirmation of the programmer's expectations.

Reading See Stoy (1977, pages 12–23) and Gunter (1992, pages 9–26) for further discussion on the various approaches to semantics.

2.11 Inadequacies

The language L_1 and the semantics we have given for it illustrate some of the important features of semantics but leave questions unanswered and problems unsolved. Issues that we have omitted include the following.

- ▷ The semantics do not allow for programs to fail. For example, if we included a division operation, how would we assign a meaning to the command $x := 1/0$? Other forms of failure include the use of non-initialized variables and non-termination.
- ▷ The language L_1 lacks many important features that we find in practical programming languages, such as declarations, types, functions, procedures, exceptions, and modules. A complete semantics must assign a meaning to all of these features.

A full discussion of these features requires additional mathematical machinery. This is the topic of the next section.

Exercise 15 Assume that the command `repeat` c `until` b is defined as in Pascal. Extend each of the semantics of Section 2 of the Notes (that is, operational, transition, denotational, natural, axiomatic, and weakest precondition) to include the `repeat` command. □

Exercise 16 Suppose that we add the Boolean operator `and` to L_1. There are two ways of evaluating P `and` Q. The first way is "strict": evaluate P, then evaluate Q, then evaluate their conjunction. The second way is "lazy": evaluate P; if P is false, return false; otherwise evaluate Q and return its value. Give denotational semantic equations for both forms of `and`, and then do the same for the Boolean operator `or`.

Discuss the relative advantages of the two modes of evaluation. □

Exercise 17 Suppose that you were deriving the axiomatic semantics of Section 2.8 from the denotational semantics of Section 2.6. Discuss how you would relate the semantic equation $\mathcal{C}\,[\![c]\!]\,\sigma = \sigma'$ to the theorem $\{\,P\,\}\ S\ \{\,Q\,\}$. Give examples of the derivation of an inference rule in the axiomatic semantics from the semantic equations. □

3 Mathematical Foundations

3.1 Useful Sets

We formally introduce some sets for subsequent use.

Definition 5 *The set* $\mathbb{T} = \{\, \mathsf{T}, \mathsf{F} \,\}$ *is the set of* **truth values***.*

Definition 6 *The set* $\mathbb{N} = \{\, 0, 1, 2, \ldots \,\}$ *is the set of* **natural numbers***.*

There are various conventions for distinguishing the sets $\{\, 0, 1, 2, \ldots \,\}$ and $\{\, 1, 2, 3, \ldots \,\}$. As stated in the definition, we will assume $0 \in \mathbb{N}$ unless otherwise stated.

Definition 7 *The set* $\mathbb{Z} = \{\, \ldots, -2, -1, 0, 1, 2, \ldots \,\}$ *is the set of* **integers***.*

We will usually use natural numbers for theoretical studies but, of course, practical languages provide *integer* as a type rather than *natural*.

Definition 8 *The set* $\mathbb{Q}$ *is the set of* **rational numbers***.*

We can think of rational numbers as pairs of integers: $(2, 3)$ denotes the rational number usually written $\frac{2}{3}$. We assume that, if (p, q) is a rational number, then $\gcd(p, q) = 1$ (thus, for instance, $(4, 6) \notin \mathbb{Q}$). Also,$(p, 0) \notin \mathbb{Q}$.

Definition 9 *The set* $\mathbb{R}$ *is the set of real numbers.*

Since programming languages do not provide real numbers, we will use them mostly in examples. The type *real* (or *float*) provided by processors and programming languages is actually a rather weird subset of the rationals.

Note 10 (Binary Operator Conventions) We will use unary and binary operators for these sets in the usual way without further definition (see Figure 15).

We will assume consistently that, if two relation symbols are mirror images of one another, the operators are inverse. Underlining a relation symbol indicates reflexive closure.

This convention applies, for instance, to the operator $\sqsubset$. It says that $x \sqsubset y$ is equivalent to $y \sqsupset x$ and that $x \sqsubseteq y$ is equivalent to $(x \sqsubset y) \vee (x = y)$. □

Type	Functions
$\mathbb{N} \times \mathbb{N} \to \mathbb{N}$	$+ \quad - \quad \times \quad \div$
$\mathbb{T} \to \mathbb{T}$	$\neg$
$\mathbb{T} \times \mathbb{T} \to \mathbb{T}$	$\wedge \quad \vee \quad \Rightarrow \quad \Longleftrightarrow$
$\mathbb{N} \times \mathbb{N} \to \mathbb{T}$	$= \quad \neq \quad < \quad \leq \quad > \quad \geq$

Figure 15: Operators for $\mathbb{N}$ and $\mathbb{T}$

3.2 Well-Founded Relations

Structural induction is a proof technique used to prove a property of a recursively-defined structure. It is a special case of a more general form of induction introduced by Noether (see box) and called "Noetherian" or "well-founded" induction.

Let S be a set and $\sqsubset$ a relation on S. Informally, we can read $a \sqsubset b$ as "a is less than b". A **descending chain** in S is a sequence of elements $x_0 \sqsupset x_1 \sqsupset \cdots \sqsupset x_n$, with $x_i \in S$.

Definition 11 (Well-Founded Relation) *The relation $\sqsubset$ on S is* **well-founded** *if it has no infinite descending chains.*

If $\sqsubset$ is a well-founded relation on a set S, we say that the pair $(S, \sqsubset)$ is a **well-founded set**.

Examples:

- ▷ The set $(\mathbb{N}, <)$ is well-founded. The longest possible descending chain starting from $n \in \mathbb{N}$ is $n > n-1 > \cdots > 1 > 0$, which is finite. All subsets of $\mathbb{N}$ have 0 as a lower bound under this ordering.
- ▷ The set $(\mathbb{N}, |)$ is well-founded. The relation $|$ is the "divides" relation: $x \mid y$ means "x divides y, so that $3 \mid 12$, $13 \mid 91$, but not $4 \mid 9$. All subsets of $\mathbb{N}$ have 1 as a lower bound under this ordering.
- ▷ The set $(\mathbb{Z}, <)$ is not well-founded because for any integer n there is an infinite descending chain $n > n-1 > n-2 > \cdots$.
- ▷ If S is a set and $\mathcal{P}(S)$ is its powerset (set of all its subsets), then $(\mathcal{P}(S), \subseteq)$ is well-founded because all subsets of $\mathcal{P}(S)$ have the empty set, $\varnothing$, as a lower bound.

If a relation has cycles, it also has infinite descending chains. Consequently, a well-founded relation must be acyclic. In particular, it must be irreflexive.

Emmy Noether (1882–1935) played an important role in the development of abstract algebra by emphasizing an axiomatic approach. She also made contributions to Galois theory, calculus of variations, rings, representation, and algebraic number theory. Her work on differential invariants contributed to general relativity. For a while, Noether was denied a lectureship at Göttingen, despite her obvious talents. This led to the following comment, attributed to Hilbert: "This is a University, not a bathing establishment".

Definition 12 (Minimal Element) *If A is set ordered by $\sqsubset$, then m is a minimal element of A iff $m \in A \wedge \forall x \sqsubset m \bullet x \notin A$.*

Definition 13 (Well-Founded Relation) *The relation $\sqsubset$ on the set S is* **well-founded** *iff every non-empty subset $A \subseteq S$ has a minimal element.*

Lemma 14 *Definitions 11 and 13 are equivalent.*

Proof Suppose that every non-empty subset of S has a minimal element. Suppose further that S has an infinite descending chain $x_0 \sqsupset x_1 \sqsupset \cdots \sqsupset x_n \sqsupset \cdots$. Then the set $\{ x_0, x_1, \ldots, x_n, \ldots \}$ has no minimal element, contradicting the assumption. Therefore, there are no infinite descending chains and $\sqsubset$ is a well-founded relation.

Suppose that $\sqsubset$ has no infinite descending chains. Let A be a non-empty subset of S and $x_0 \in A$. Construct a chain $x_0 \sqsupset x_1 \sqsupset \cdots \sqsupset x_n$ in A. Attempt to extend the chain: either there is $y \in A$ such that $x_n \sqsupset y$ or there is no such y. If there is such a y, extend the chain with $x_{n+1} = y$. Otherwise, x_n is a minimal element of A. □

Exercise 18 Show that the reflexive, transitive closure of a well-founded relation is a partial order. □

Theorem 15 (Noetherian Induction) *Let S be a set, $\sqsubset$ a well-founded relation on S, and P a predicate on S. (That is, if $x \in S$ then $P(x)$ is either true or false.) Then*

$$\forall x \in S \bullet P(x) \qquad \Longleftrightarrow \qquad \forall x \in S \bullet (\forall y \sqsubset x \bullet P(y)) \Rightarrow P(x).$$

In other words, we can say that to prove $P(x)$ for all $x \in S$, it is sufficient to show that:

> For all $x \in S$, if P is true for all predecessors of x, it is also true of x.

Proof The forward implication ($\Rightarrow$) is obvious. For the reverse implication ($\Leftarrow$), suppose we have proved that, for every x in S, if $P(y)$ for all predecessors y of x (that is, $y \sqsubset x$), then $P(x)$.

If x has no predecessors, then we must have proved $P(x)$. If x does have a predecessor, $x_n \sqsubset x$, we must have proved $x_n \Rightarrow x$. In fact, we must have proved $x_0 \Rightarrow x_1 \wedge x_1 \Rightarrow x_2 \wedge \cdots \wedge x_n \Rightarrow x$, where x_0 has no predecessors.

The only elements of S that might remain unproved are members of infinite descending chains. But, by hypothesis, there are no infinite descending chains. Consequently, $P(x)$ is true for all $x \in S$. □

Noetherian induction has many applications. Consider again the syntax of binary numerals, ignoring addition:

$$B = \mathsf{0} \mid \mathsf{1} \mid B\mathsf{0} \mid B\mathsf{1}$$

If x and y are binary numerals, we can define $x \sqsubset y$ if x is a substring of y. (If the grammar was more complex, we would define $x \sqsubset y$ to mean "x is a syntactic component of y".) Clearly, any descending chain in this ordering is bounded below by the empty string. Thus $\sqsubset$ is a well-founded relation and Noetherian induction allows us to prove a property of all binary numerals by proving it for each case of the syntax. When we prove the property for $B0$, for instance, we may assume that it is true for B.

Lemma 16 *Let* $\operatorname{len} x$ *denote the length of the binary numeral* x*. For all binary numerals* x*:* $\mathcal{M}[\![x]\!] \leq 2^{\operatorname{len} x} - 1$.

Proof We consider each syntactic alternative in turn. The cases `0` and `1` are straightforward.

$$\begin{aligned}\mathcal{M}[\![\mathtt{0}]\!] = 0 &\leq 1 = 2^{\operatorname{len}\mathtt{0}} - 1\\ \mathcal{M}[\![\mathtt{1}]\!] = 1 &\leq 1 = 2^{\operatorname{len}\mathtt{1}} - 1\end{aligned}$$

The other two cases are slightly more complicated. IH indicates the use of the induction hypothesis.

$$\begin{aligned}\mathcal{M}[\![x\,\mathtt{0}]\!] &= 2\mathcal{M}[\![x]\!]\\ &\leq 2(2^{\operatorname{len} x} - 1) && (IH)\\ &= 2^{\operatorname{len} x+1} - 2\\ &\leq 2^{\operatorname{len} x\mathtt{0}} - 1\\ \mathcal{M}[\![x\,\mathtt{1}]\!] &= 2\mathcal{M}[\![x]\!] + 1\\ &\leq 2(2^{\operatorname{len} x} - 1) + 1 && (IH)\\ &= 2^{\operatorname{len} x+1} - 1\\ &= 2^{\operatorname{len} x\mathtt{1}} - 1\end{aligned}$$

The result follows by the principle of Noetherian induction. □

Exercise 19 Attempt to prove the weaker result $\mathcal{M}[\![x]\!] \leq 2^{\operatorname{len} x}$ using induction, as above. The lesson of this exercise is that an induction hypothesis may need strengthening to complete a proof. □

Reading Winskel (1993, Chapter 3) provides a detailed discussion of induction principles.

3.3 Recursive Definitions

In mathematics in general, and semantics in particular, recursive functions play an important role. We have seen that we need recursive functions to explain even such simple constructs as the `while` command (see (43) on page 11). We note further that non-recursive functions are not very interesting. If, for example, we define

$$f(x) \quad \triangleq \quad x^2 + 1$$

we can simply replace $f(E)$, wherever it occurs, by $(x^2+1)[E/x]$.

A recursive definition has the form

$$x \quad \triangleq \quad \cdots x \cdots.$$

We can view this definition as an equation

$$x \quad = \quad \cdots x \cdots \qquad (75)$$

for which we seek a solution for x. This may be easy: if the equation is $x = 2x - 1$, the obvious solution is $x = 1$. Equations may also be useless: $x = x$ is satisfied by all values of x; or insoluble: $x = x + 1$ is not satisfied by any values of x.

We can write the recursive equation (75) as

$$x \quad = \quad \phi(x) \tag{76}$$

which shows that solving recursive equations is equivalent to finding fixed points: a *fixed point* of a function ϕ is simply a value x such that $\phi(x) = x$.

Recursion of the form (75) is usually called *immediate recursion.* We are often more interested in the recursive definition of functions. The definition of the function will look like this:

$$f(x) \quad \triangleq \quad \cdots\ f\ \cdots \tag{77}$$

and we can rewrite this as an equation

$$f \quad = \quad \lambda x\,.\ \cdots\ f\ \cdots \quad = \quad H(f) \tag{78}$$

so that any fixed point of H is a solution of (78). To use a familiar example, the factorial function is a fixed point of H where

$$H(f) \quad = \quad \lambda x\,.\,\texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x \times f(x-1).$$

There is another difficulty that at first sight seems unrelated to recursion: there are too many functions. Since the number of functions from a set A to a set B is $|B|^{|A|}$, the number of functions from $\mathbb{N}$ to $\mathbb{N}$ is ω^ω, which is uncountable. But the number of functions that we can describe (that is, write programs) is countable. In other words, most functions cannot be described by computations are therefore uncomputable.

Strachey (see box) was the first to recognize the importance of recursive functions in computer science. He decided to use λ-calculus to formalize programming in terms of recursive function theory.

3.4 λ-Calculus

This section provides a brief review of the salient features of λ-calculus: for details, see Stoy (1977, Chapter 5).

Any theory of functions must begin by solving a simple but important problem: how do we write the definition of a function? The conventional notation, for example $f(x) = x^2 + 1$, is unsatisfactory because x appears on both the left and the right side. We cannot simply omit x on the left side, because this would make definitions ambiguous (is $f = x^2 + y^2$ a function of x, y, or both?). Church's solution to this problem was λ-*notation*: we write

$$f \quad = \quad \lambda x\,.\,x^2 + y^2$$

Christopher Strachey (1916–1975), a nephew of Lytton Strachey, laid the foundations for programming language semantics. His career in computer science began in 1950 with a program for playing checkers. During the early fifties, he programmed simulations of water flows for the St. Lawrence Seaway, then under construction. During the sixties, Strachey became interested in the precise definition of programming languages and began using the λ-calculus to formulate his definitions.

which makes it clear that f is a function of x only and that y is a free variable. If we wanted f to depend on both x and y, we would write instead

$$f \quad = \quad \lambda x \,.\, \lambda y \,.\, x^2 + y^2$$

From this definition, Church constructed a *calculus of functions*, now known simply as the λ-*calculus*. The syntax of the λ-calculus is very simple: it consists of formulas M defined by

$$M \quad = \quad x \mid MM \mid \lambda x \,.\, M \mid (M)$$

in which x is a *variable*; MM is an *application* (in λ-calculus, $f(x)$ is written fx); $\lambda x \,.\, M$ is an *abstraction*; and (M) indicates that we can use parentheses in the usual way.

In the syntax, λ is a *binding operator*, analogous to $\forall$, $\exists$, Σ, or $\int$. In the term xy of the formula $\lambda x \,.\, xy$, x occurs *bound* and y occurs *free*. We are allowed to change the names of bound variables consistently: for example, $\lambda x \,.\, xy = \lambda z \,.\, zy$ (compare $\forall x \,.\, P(x) \equiv \forall y \,.\, P(y)$).

Note 17 (Syntactic Conventions for λ-calculus) Application is left associative: LMN means $(LM)N$ and not $L(MN)$.

The body of a λ-abstraction extends as far as possible to the right. For example, $\lambda x \,.\, xyz$ means $\lambda x \,.\, (xyz)$. If we do not want yz to be part of the function body, we must write $(\lambda x \,.\, x)yz$.

As usual, we write $M[N/x]$ to mean "the expression M, with all occurrences of x replaced by N".

The following rules of the λ-calculus look fairly simple. Read $\leftrightarrow$ as "can be replaced by".

$$\begin{array}{lrcl}
(\alpha)\ \text{If } y \text{ is not free in } M\text{, then} & \lambda x \,.\, M & \leftrightarrow & \lambda y \,.\, M[y/x] \\
(\beta) & (\lambda x \,.\, M)N & \leftrightarrow & M[N/x] \\
(\eta)\ \text{If } x \text{ is not free in } M\text{, then} & \lambda x \,.\, Mx & \leftrightarrow & M
\end{array}$$

Rule (α) simply repeats what we said above: bound names can be changed. Rule (β) is the important one, because it shows how functions are applied to arguments.

The rules for bound variables and substitution involve some subtleties: see Stoy (1977, pages 59 and 62).

The λ-calculus has a *fixed point operator*, Y, with the following curious property: for any function f, we have $f(Yf) = f$ (that is, Yf is a fixed point of f). The fixed point operator does not cause problems in the pure λ-calculus, but Haskell Curry showed that it rendered any application of λ-calculus inconsistent.

Suppose that we attempt to combine λ-calculus with propositional calculus (axioms 1 and 2 below). Then we can prove the truth of an arbitrary proposition q as follows. Define the function f by

$$f \quad = \quad \lambda x \,.\, x \Rightarrow q$$

We begin by constructing a fixed point of f. Let $g = \lambda y \,.\, f(yy)$. Then

$$\begin{aligned}
gg &= (\lambda y \,.\, f(yy))(\lambda y \,.\, f(yy)) \\
&= f(\lambda y \,.\, f(yy))(\lambda y \,.\, f(yy)) \\
&= f(gg)
\end{aligned}$$

Let $p = gg$, so that $f(p) = p$. The formal proof then continues as follows.

1	$P \Rightarrow P$	(Axiom)
2	$(P \Rightarrow (P \Rightarrow Q)) \Rightarrow (P \Rightarrow Q)$	(Axiom)
3	$f = \lambda x \,.\, x \Rightarrow q$	(Definition of f)
4	$f(p) = p$	(See above)
5	$p = (p \Rightarrow q)$	(3,4)
6	$p \Rightarrow p$	(1)
7	$p \Rightarrow (p \Rightarrow q)$	(5,6)
8	$p \Rightarrow q$	(2,7)
9	p	(5,8)
10	q	(8,9)

Curry's paradox put λ-calculus into disrepute amongst mathematicians. Dana Scott's contribution to computer science was to construct a model of the λ-calculus that excludes non-computable functions yet provides solutions to all equations of the form (78).

Exercise 20 The purpose of this exercise is to provide practice in the manipulation of λ expressions.

λ-calculus has been used (rather unsuccessfully) as the foundation of mathematics. The first thing we need in mathematics is the natural numbers. We can define the "Church numerals" $\{\,\overline{0}, \overline{1}, \overline{2}, \ldots\,\}$ as follows (read $\sim$ as "represents"):

$$\begin{array}{rcll}
\overline{0} & = & \lambda x \lambda y \,.\, y & \sim \; 0 \\
\overline{1} & = & \lambda x \lambda y \,.\, xy & \sim \; 1 \\
\overline{2} & = & \lambda x \lambda y \,.\, x(xy) & \sim \; 2 \\
\overline{3} & = & \lambda x \lambda y \,.\, x(x(xy)) & \sim \; 3 \\
& \ldots & & \\
\overline{n} & = & \lambda x \lambda y \,.\, \underbrace{x(x(\cdots(x}_{n} y)\cdots) & \sim \; n
\end{array}$$

The function $S \equiv \lambda x \lambda y \lambda z \,.\, y(xyz)$ is the successor function. Show that $S\overline{0} = \overline{1}$ and $S\overline{1} = \overline{2}$.

We also need a form of conditional. Define these functions:

$$\begin{array}{rcl}
T & \equiv & \lambda x \lambda y \,.\, x \\
F & \equiv & \lambda x \lambda y \,.\, y
\end{array}$$

and show that $Txy \to x$ and $Fxy \to y$. We use T to represent "true"and F to represent "false".

Using the conditional functions, we can define

$$Z \;\equiv\; \lambda x \,.\, x(T(F))(T).$$

Show that $Z\overline{n} \to T$ if $\overline{n} = \overline{0}$ and $Z\overline{n} \to F$ otherwise. □

3.5 Lattices

Suppose that S is a set with a partial order $\sqsubseteq$. If S is small, we can display it as a Hasse diagram, as in Figure 16. An edge between two vertices in a Hasse diagram indicates that they are related by the partial order. Transitive edges are not shown. If $x \sqsubseteq y$, then y should appear higher on the page than x.

Definition 18 *An element $u \in S$ is an* **upper bound** *of a subset $A \subseteq S$ if $x \in A \Rightarrow x \sqsubseteq u$. If the set of upper bounds of A has a least element, that element is a* **least upper bound** *of A.*

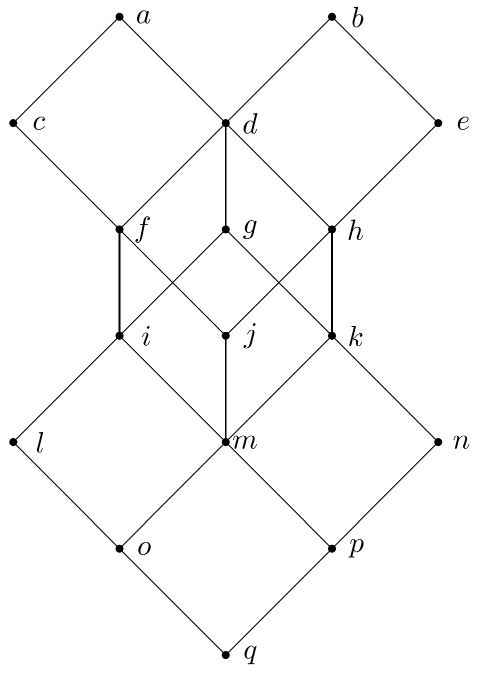

Figure 16: A Hasse diagram

In Figure 16, $f \sqsubseteq b$ and $q \sqsubseteq a$. The elements a, b, d, and g are upper bounds of the set $\{ g, i, k, m \}$. The elements c, f, e, h, i, and k are *not* upper bounds of this set. The least upper bound of $\{ g, i, k, m \}$ is g. The least upper bound of a set does not have to be a member of the set: g is also the least upper bound of $\{ i, k, m \}$. Some sets, such as $\{ a, b, c, d, e \}$ do not have upper bounds.

Lemma 19 *Let S be a set ordered by $\sqsubseteq$ with a subset A. If A has a least upper bound, the least upper bound is unique.*

Proof Let u and v be least upper bounds of A. Since u is minimal in the set of upper bounds, $u \sqsubseteq v$. Similarly, $v \sqsubseteq u$. Hence $u = v$. □

We will write $x \sqcup y$ for the least upper bound of the set $\{ x, y \}$, if it exists, and $\bigsqcup A$ for the least upper bound of the set A, if it exists. Similarly, we define *lower bound*, and *greatest lower bound*; and we write $x \sqcap y$ for the greatest lower bound of x and y and $\bigsqcap A$ for the greatest lower bound of the set A.

The operations $\sqcup$ and $\sqcap$ are called *join* and *meet* respectively.

Definition 20 *Let S be a set partially ordered by $\sqsubseteq$.*

1. *$(S, \sqsubseteq)$ is a* **lattice** *iff $x \sqcup y$ and $x \sqcap y$ both exist for all $x, y \in S$.*
2. *$(S, \sqsubseteq)$ is a* **complete lattice** *if $\bigsqcup A$ and $\bigsqcap A$ both exist for all $A \subseteq S$.*

If $(S, \sqsubseteq)$ is a complete lattice, it has a *top element* $\top = \bigsqcup S = \bigsqcap \varnothing$ and a *bottom element* $\bot = \bigsqcap S = \bigsqcup \varnothing$. $\top$ and $\bot$ are pronounced "top" and "bottom", respectively. The structure in Figure 16 is not a lattice because, for example, there is no element $a \sqcup b$.

The lattice operators obey various laws, as shown below.

$$\begin{array}{llcllcl}
\text{Idempotence:} & x \sqcup x & = & x & x \sqcap x & = & x \\
\text{Commutativity:} & x \sqcup y & = & y \sqcup x & x \sqcap y & = & y \sqcap x \\
\text{Associativity:} & x \sqcup (y \sqcup z) & = & (x \sqcup y) \sqcup z & x \sqcap (y \sqcap z) & = & (x \sqcap y) \sqcap z \\
\text{Absorption:} & x \sqsubseteq y & \Leftrightarrow & x \sqcup y = y & x \sqsubseteq y & \Leftrightarrow & x \sqcap y = x \\
\text{Distributivity:} & x \sqcup (y \sqcap z) & = & (x \sqcup y) \sqcap (x \sqcup z) & x \sqcap (y \sqcup z) & = & (x \sqcap y) \sqcup (x \sqcap z)
\end{array}$$

The distributive law does not hold in all lattices. If it does, we way that the lattice is *distributive* or *modular*. We will not need distributivity for semantics.

We may be interested in joins but not meets, or *vice versa*. A structure with one operation but not the other defined is called a *semi-lattice*.

3.5.1 Examples of Lattices

- ▷ $(\mathbb{N}, \le)$ is a lattice with $x \sqcup y = \max(x, y)$ and $x \sqcap y = \min(x, y)$. We have $\bigsqcap \mathbb{N} = 0$ but, since $\bigsqcup \mathbb{N}$ does not exist, the lattice is not complete.
- ▷ $(\mathbb{Q}, \le)$ is also a lattice with $x \sqcup y = \max(x, y)$ and $x \sqcap y = \min(x, y)$. It is not complete because it contains sets such as $\{\, x \in \mathbb{Q} \mid x^2 < 2 \,\}$ that have no least upper bound. (The upper bound of this set is the real, irrational number $\sqrt{2}$. Dedekind (1831–1916) was the first person to define real numbers as least upper bounds of sets of rational numbers.)
- ▷ $(\mathbb{N}, \mid)$, in which $\mid$ is the divisibility relation, is a lattice with $x \sqcup y = \operatorname{lcm}(x, y)$ and $x \sqcap y = \gcd(x, y)$. ("lcm" and "gcd" stand for "least common multiple" and "greatest common divisor", respectively.) The divisibility relation $x \mid y$ (read: x divides y) with $x, y \in \mathbb{N}$ is defined by
$$x \mid y \iff \exists k \in \mathbb{N} \,.\, kx = y.$$
- ▷ For any set S, the set of all subsets of S, $\mathcal{P}(S)$, with the subset ordering $\subseteq$, is a lattice. The join and meet of $A, B \in \mathcal{P}(S)$ are $A \cup B$ and $A \cap B$, respectively. The lattice is complete with bottom $\varnothing$ and top S.
- ▷ The set of propositions ordered by implication forms a complete lattice with $\vee$ as the join operation, $\wedge$ as the meet operation, `true` as the top element, and `false` as the bottom element. Figure 17 shows a sublattice of this lattice that contains propositions with two proposition symbols.
- ▷ From any set S, we can create a *flat lattice* $S_\bot$ by adding a bottom element, $\bot$, and defining an order $\sqsubseteq$ like this:
$$\forall x, y \in S \,.\, x \sqsubseteq y \iff x = \bot.$$

For example, Figure 18 on page 32 shows the lattice $\mathbb{N}_\perp = \mathbb{N} \cup \{ \perp \}$, in which natural numbers are incomparable but every number is greater than $\perp$.

▷ We can consider an interval $[a, b]$ with $a, b \in \mathbb{R}$ and $a \le b$ to be an approximation to a real number. For example, $[3.141, 3.142]$ approximates π. We define the relation $\sqsubseteq$ on intervals by

$$[a, b] \sqsubseteq [c, d] \quad \iff \quad a \le c \wedge d \le b$$

and say that, if $[a, b] \sqsubseteq [c, d]$, then $[c, d]$ is a *better approximation* than $[a, b]$.

Intervals form a semi-lattice with bottom $[-\infty, \infty]$ and meet operation

$$[a, b] \sqcap [c, d] \quad = \quad [\min(a, c), \max(b, d)]$$

The maximum elements of this lattice are intervals $[x, x]$ with $x \in \mathbb{R}$ and there is no upper bound.

If we remove the constraint $a \le b$ in the interval $[a, b]$, we obtain intervals with "too much" information. The set of all intervals forms a complete lattice with top element $[\infty, -\infty]$ and join operation

$$[a, b] \sqcup [c, d] \quad = \quad [\max(a, c), \min(b, d)].$$

▷ We can define an *implementation* relation on specifications and programs: $P \sqsubseteq Q$ means "P implements Q". The program $P \sqcup P'$ is a program that implements either P or P' and the program $P \sqcap P'$ is a program that implements both P and P'. For example, if P is "permute" and P' is "put in order", then $P \sqcap P'$ is "sort". The bottom of the lattice is a useless specification that requires nothing and the top of the lattice is an omnipotent program that satisfies all specifications.

▷ In some object oriented languages with multiple inheritance, classes form a semi-lattice ordered by inheritance. The top element is a class such as *Object* in Smalltalk or *Any* in Eiffel. If C and C' are classes, their join $C \sqcup C'$ is their closest common ancestor and their meet $C \sqcap C'$, which does not necessarily exist, is their closest common descendant. Some languages define a bottom element (making the lattice complete): the bottom element is a class that can accept all messages.

▷ Let L be the set of linear subspaces of a three-dimensional vector space. That is, L consists of the empty set, points, lines, planes, and the entire space. We order the components of L by inclusion: $x \sqsubseteq y$ iff every point of x is also a point of y. Thus a line may be included in a plane, for example.

Define $x \sqcup y$ as the smallest linear subspace that contains both x and y. For example, if x and y are lines, $x \sqcup y$ might be a line (if x and y are the same line), a plane (if x and y intersect), or the entire space (otherwise). Define $x \sqcap y$ as the set-wise intersection of x and y (the intersection is always a linear subspace). Then $(L, \sqsubseteq)$ is a lattice with the empty set as its bottom element and the entire space as its top element.

Exercise 21 Is the lattice $(\mathbb{N}, |)$ complete? □

Exercise 22 Verify the lattice laws for the interval lattice defined above. □

3.5.2 Functions on a Lattice

Suppose that $(S, \sqsubseteq)$ is a lattice and $f : S \to S$ is a function.

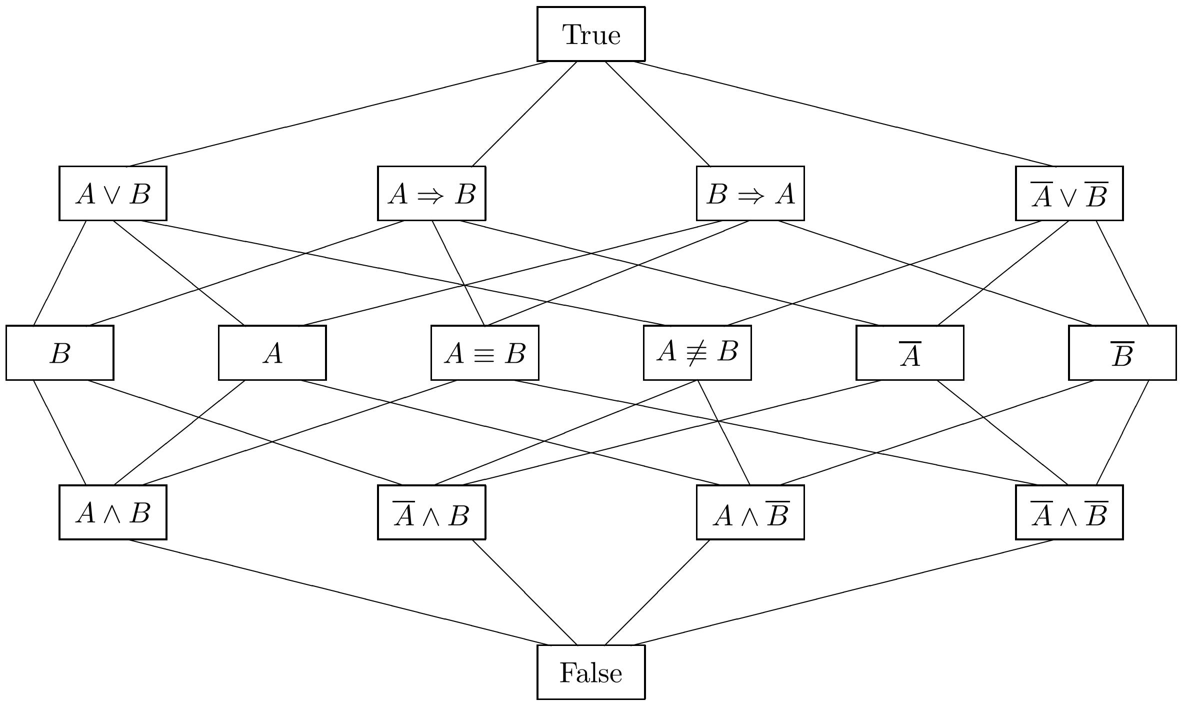

Figure 17: Lattice of two-place propositions

Definition 21 (Monotonicity) *The function f is* **monotonic** *if it preserves the lattice ordering:*

$$\forall x, y \in S \cdot x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y).$$

In conventional analysis, a function $f : \mathbb{R} \to \mathbb{R}$ is monotonic if $x \le y \Rightarrow f(x) \le f(y)$. There is an obvious analogy with monotonic functions on a lattice.

Definition 22 (Directed Set) *The set $X \subseteq S$ is* **directed** *if every finite subset of X has an upper bound in X.*

In Figure 16, the set $X = \{\, i, j, k, l, m, n \,\}$ is not directed because its least upper bound is $d \notin X$.

In practice, the directed sets that we are interested in are almost always chains. In Figure 16, for example, there are several ascending chains, including

$$q \sqsubseteq p \sqsubseteq n \sqsubseteq k \sqsubseteq g \sqsubseteq d \sqsubseteq a.$$

The members of an ascending chain obviously form a directed set.

Definition 23 (Continuity) *The function f is* **continuous** *if it preserves upper bounds of the lattice: for every directed set $X \subseteq S$,*

$$f(\bigsqcup X) \quad = \quad \bigsqcup f(X) \quad = \quad \bigsqcup \{\, f(x) \mid x \in X \,\}.$$

Most reasonable functions are continuous. The intuitive idea of continuity is most easily understood by considering functions that are not continuous. Let $f : \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$ be defined by

$$f(S) = \begin{cases} \varnothing, & \text{if } S \text{ is finite;} \\ \mathbb{N}, & \text{otherwise.} \end{cases}$$

and let $\mathcal{F}$ be the set of all finite subsets of $\mathbb{N}$. Then $f(S) = \varnothing$ for every $S \in \mathcal{F}$ and so $\bigcup f(\mathcal{F}) = \varnothing$. But $\bigcup \mathcal{F} = \mathbb{N}$ and so $f(\bigcup \mathcal{F}) = \mathbb{N}$. Thus $\bigcup f(\mathcal{F}) \neq f(\bigcup \mathcal{F})$ and f is not continuous.

Lemma 24 *If the function f is continuous, it is also monotonic.*

Proof Suppose that f is continuous and $x \sqsubseteq y$. Then:

$$\begin{aligned} f(y) &= f(x \sqcup y) && \text{(absorption)} \\ &= f(x) \sqcup f(y) && \text{(continuity)} \\ f(x) &\sqsubseteq f(y) && \text{(absorption)} \end{aligned}$$

□

For most purposes, it is sufficient to know that a function is monotonic. The distinction between monotonicity and continuity is important only in a few situations where we need to ensure that the theory is complete. See, for example, Winskel (1993, page 122).

Tarski showed that any continuous function on a lattice has fixed points; in particular, there is a fixed point that is minimal under the lattice ordering.

Definition 25 *We define "powers" of a function as follows:*

$$\begin{aligned} f^0(x) &= x, \\ f^n(x) &= f(f^{n-1}(x)) && \textit{for } n \geq 1. \end{aligned}$$

Theorem 26 (Tarski) *If f is a continuous function on a lattice $(S, \sqsubseteq)$, it has a least fixed point, p, given by*

$$p = \bigsqcup_{n=0}^{\infty} f^n(\bot).$$

Proof We show first that p is a fixed point of f.

$$\begin{aligned} f(p) &= f\left(\bigsqcup_{n=0}^{\infty} f^n(\bot)\right) \\ &= \bigsqcup_{n=0}^{\infty} f(f^n(\bot)) && \text{(continuity)} \\ &= \bigsqcup_{n=0}^{\infty} f^{n+1}(\bot) \sqcup \{\,\bot\,\} && \text{(monotonicity)} \\ &= \bigsqcup_{n=0}^{\infty} f^n(\bot) \\ &= p \end{aligned}$$

Next, we show that p is the least fixed point. Suppose that q is a fixed point of f. Then $f(q) = q$. Since f is monotonic, $f(\bot) \sqsubseteq f(q) = q$. By induction, $f^n(\bot) \sqsubseteq q$ and therefore $\bigsqcup_{n=0}^{\infty} f^n(\bot) \sqsubseteq q$, again by monotonicity. Thus $p \sqsubseteq q$. $\Box$

If we use ascending chains rather than directed sets, we can write Tarski's result in a slightly simpler form. For an ascending chain $a_1 \sqsubseteq a_2 \sqsubseteq \cdots \sqsubseteq a_n$, we have $\bigsqcup\{\, a_1, a_2, \ldots, a_n \,\} = a_n$ and $\bigsqcup_{n=0}^{\infty} f^n(\bot)$ is simply $f^\infty(\bot)$.

In a finite lattice, an ascending chain must eventually become "stationary":

$$a_1 \sqsubseteq a_2 \sqsubseteq \cdots \sqsubseteq a \sqsubseteq a \sqsubseteq \cdots.$$

Here are some examples of least fixed points on the lattices we have seen.

- ▷ Consider the division lattice on the natural numbers, $(\mathbb{N}, |)$, and let $f(x) = x^k$ for some $k \geq 0$. Then f is monotonic because $x \mid y \Rightarrow x^k \mid y^k$. By Tarski's theorem, the least fixed point of f is $f^\infty(1) = 1$.
- ▷ Consider the lattice of real intervals with typical element $[x, y]$ and define the function f by

$$f[x,y] \quad = \quad \left[\frac{9x+y}{10}, \frac{x+9y}{10}\right].$$

  It is easy to see that f is monotonic. By Tarski's theorem, its least fixed point is $f^\infty[-\infty, \infty] = [-\infty, \infty]$. Note that $[x, x]$ is a fixed point of f for every x, but not the *least* fixed point.
- ▷ Let $\mathcal{R}$ be a set of rules of the form $\dfrac{X}{y}$ where X is a set of premises and y is a conclusion. (In the following, we write rules as $X \vdash y$ to save space.) We write axioms as $\varnothing \vdash y$. Define the function R on sets of facts by

$$R(A) \quad = \quad \{\, y \mid \exists X \subseteq A \,.\, X \vdash y \in \mathcal{R} \,\}.$$

  Thus $R(A)$ is the set of all facts that can be derived from the set of facts A using the rules $\mathcal{R}$. For example, $R(\varnothing)$ is the set of axioms of $\mathcal{R}$.

  Clearly, R is monotonic under the subset ordering. Since the set of rules is finite, it is also continuous. Thus it has a least fixed point

$$S \quad = \quad R^\infty(\varnothing).$$

  S is the smallest set of facts that it is closed under $\mathcal{R}$. In other words, it is precisely the set of facts that we can infer using $\mathcal{R}$. (See also Winskel (1993, pages 52–3).)

Tarksi's theorem is all that we need for semantics, but it is actually only a part of the story. In fact, the set of fixed points of a continuous function on a lattice formas a sublattice with the least fixed point as its bottom element and the greatest fixed point as its top element.

3.6 Domains

A *domain* D is a lattice with an ordering $\sqsubseteq$ called the **information ordering**. Informally, if $x, y \in D$ and $x \sqsubseteq y$, then we know more about y than we know about x. Alternatively, we need more computation to obtain y than to obtain x. All domains have a bottom element, $\perp$, which corresponds to "no information". Some domains have a top element, $\top$, which corresponds to "too much information".

The simplest domains are **flat**, or **discrete**, domains. Elements are unrelated except that $\perp \sqsubseteq x$ for every x. Figure 18 shows the flat lattice of natural numbers, $\mathbb{N}_\perp$.

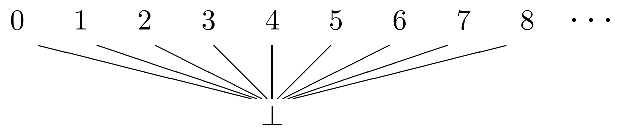

Figure 18: The Lattice $\mathbb{N}_\perp$

Exercise 23 Draw the lattice $\mathbb{T}_\perp$. $\square$

Given two domains D_1 and D_2 we can construction sum and product domains.

3.6.1 Sum Domains

Elements of the sum domain, $D_1 + D_2$, are pairs (i, x) where $i \in \{1, 2\}$ and $x \in D_1$ if $i = 1$ or $x \in D_2$ if $i = 2$. (A sum domain corresponds to a tagged variant record in Pascal or a tagged union in C.) The ordering is defined by

$$(i, x) \sqsubseteq (j, y) \iff i = j \wedge x \sqsubseteq y.$$

We have the choice of adding a new bottom element, creating a *separated sum*, or of combining the bottom elements, creating a *coalesced sum*. Figure 19 shows both forms of $\mathbb{T}_\perp + \mathbb{T}_\perp$. The separated sum is usually the most convenient form to use.

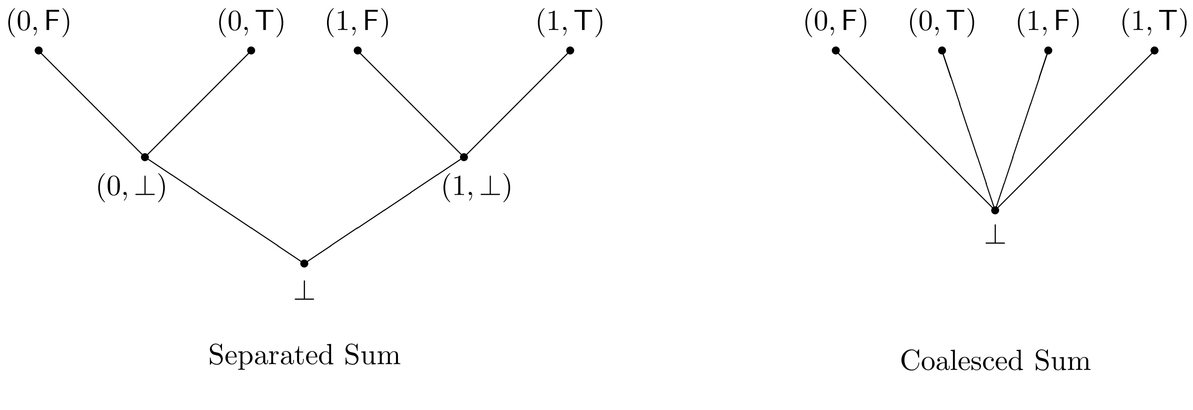

Figure 19: Different forms of $\mathbb{T}_\perp + \mathbb{T}_\perp$

3.6.2 Product Domains

Elements of the product domain, $D_1 \times D_2$, are pairs (x_1, x_2) with $x_1 \in D_1$ and $x_2 \in D_2$. The ordering is defined by

$$(x_1, x_2) \sqsubseteq (y_1, y_2) \iff x_1 \sqsubseteq y_1 \wedge x_2 \sqsubseteq y_2.$$

Figure 20 shows the product domain $\mathbb{T}_\perp \times \mathbb{T}_\perp$; note how the information increases from the bottom to the top of the diagram. It is straightforward to prove that, if D_1 and D_2 are domains, then $D_1 + D_2$ and $D_1 \times D_2$ really are domains.

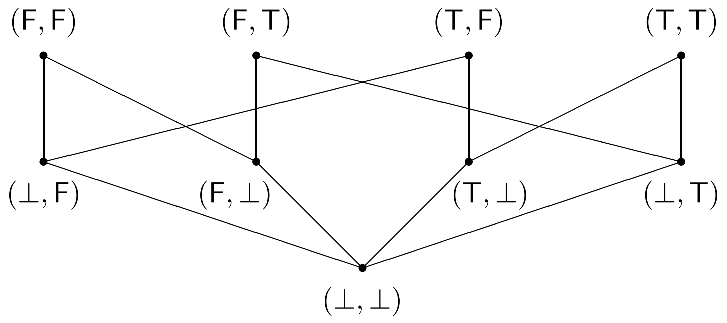

Figure 20: The Lattice $\mathbb{T}_\perp \times \mathbb{T}_\perp$

An alternative ordering for the product domain would be *lexicographic ordering*:

$$(x_1, x_2) \sqsubseteq (y_1, y_2) \iff x_1 \sqsubseteq y_1 \vee x_1 = y_1 \wedge x_2 \sqsubseteq y_2.$$

Exercise 24 If A is a finite set with n members, how many members does $A_\perp \times A_\perp$ have? $\square$

3.6.3 Function Domains

Of greater interest is the fact that we can construct a function domain from any two domains. The function domain is written $D_1 \to D_2$ and it consists of all continuous functions, ordered pointwise. That is, if $f : D_1 \to D_2$ and $g : D_1 \to D_2$, then

$$f \sqsubseteq g \iff \forall x \in D_1 \,.\, f(x) \sqsubseteq g(x).$$

The functions in this domain are precisely the functions that we can compute. The least function in this ordering is the function that yields $\perp$ for every value of its argument.

Consider the function domain $\mathbb{T}_\perp \to \mathbb{T}_\perp$. A function such as

$$f = \{ (\mathsf{T}, \perp), (\mathsf{F}, \mathsf{T}), (\perp, \mathsf{F}) \}$$

is not allowed because it is not monotonic: $\perp \sqsubseteq \mathsf{T}$ but $f(\mathsf{T}) \sqsubseteq f(\perp)$. Figure 21 shows the monotonic functions in $\mathbb{T}_\perp \to \mathbb{T}_\perp$. To avoid cluttering the diagram, we write p, q, r to stand for the function $\{ (\mathsf{T}, p), (\mathsf{F}, q), (\perp, r) \}$. Again, note that information increases upwards in the diagram.

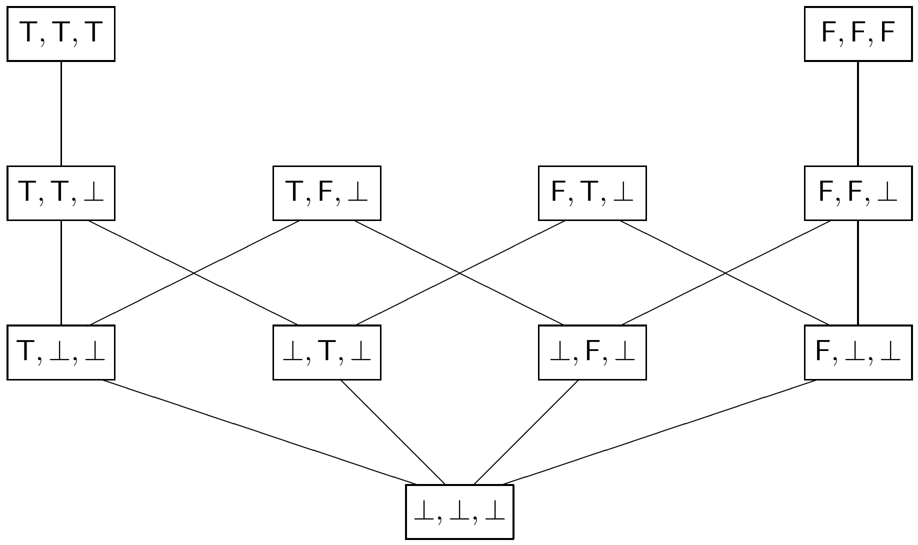

Figure 21: Monotonic functions in $\mathbb{T}_\perp \to \mathbb{T}_\perp$

In general, if D_1 and D_2 are flat domains, and $f, g : D_1 \to D_2$, then $f \sqsubseteq g$ means that either $f(x) = g(x)$ or that $f(x) = \perp$ and $g(x) \neq \perp$. Informally, we know the value of g at more points than we know the value of f. Thus the ordering corresponds to how much computation we have done. If $f \sqsubseteq g$, then f requires less computation than g.

Any partial function $f : A \to B$ (where A and B are sets) can be extended to a total function $f_\perp : A_\perp \to B_\perp$ as follows:

$$f_\perp(x) \quad = \quad \begin{cases} \perp, & \text{if } x = \perp; \\ f(x), & \text{if } x \in A; \\ \perp, & \text{otherwise.} \end{cases}$$

Thus we assume that all functions in a domain are total.

We adopt similar conventions for functions of more than one argument. In $\mathbb{N}_\perp$, for example, we assume $n + \perp = \perp + n = \perp$ and similarly for the other operators. In $\mathbb{N}$, $\div : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is a partial function, because $1 \div 0$ is undefined. The extended function, $\div_\perp : \mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathbb{N}_\perp$ is total with $1 \div 0 = \perp$.

Note 27 The standard approach uses $\perp$ and orders functions pointwise. Meyer (1990, §8.8) provides a different, non-standard theory which avoids bottoms and uses the "natural" order on functions: $f \sqsubseteq g \iff f \subseteq g$, with functions as sets of pairs. $\Box$

3.6.4 Recursive Equations and Fixed Points

We can now address the problem that we started with (on page 23): that of giving a meaning to recursive equations. We use the familiar factorial function as an example. The recursive definition can be written in λ notation as

$$f \quad = \quad \lambda x \,.\, \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x \times f(x-1).$$

Define the function H by

$$H(f) \quad = \quad \lambda x \,.\, \texttt{if } x = 0 \texttt{ then } 1 \texttt{ else } x \times f(x-1).$$

Then we require $f = H(f)$; in other words, the factorial function should be the least fixed point of H. By Tarski's theorem, the required fixed point is $H^{\infty}(\bot_{\mathbb{N}\to\mathbb{N}})$. To see how this works, we consider the following sequence of functions:

$$\begin{array}{lcl}
H^0(\bot) & = & \bot \\
H^1(\bot) & = & \lambda x \,.\, \texttt{if } x = 0 \texttt{ then } 1 \texttt{ else } x \times \bot(x-1) \\
H^2(\bot) & = & \lambda x \,.\, \texttt{if } x = 0 \texttt{ then } 1 \texttt{ else } x \times H^1(\bot)(x-1) \\
\ldots\ldots & & \\
H^n(\bot) & = & \lambda x \,.\, \texttt{if } x = 0 \texttt{ then } 1 \texttt{ else } x \times H^{n-1}(\bot)(x-1)
\end{array}$$

Evaluating these gives:

$$\begin{array}{rcl}
H^0(\bot) & = & \{\,(0,\bot),(1,\bot),(2,\bot),\ldots\,\} \\
H^1(\bot) & = & \{\,(0,1),(1,\bot),(2,\bot),\ldots\,\} \\
H^2(\bot) & = & \{\,(0,1),(1,1),(2,\bot),\ldots\,\} \\
\ldots\ldots & & \\
H^k(\bot)(x) & = & \begin{cases} x!, & \text{if } x < k; \\ \bot, & \text{if } x \geq k. \end{cases}
\end{array}$$

We have defined an ascending chain $H^0(\bot) \sqsubseteq H^1(\bot) \sqsubseteq H^2(\bot) \sqsubseteq \cdots \sqsubseteq H^k(\bot) \sqsubseteq \cdots$. In the limit, $H^{\infty}(\bot)$, computes $x!$ correctly for all finite values of x, as required.

This construction will give a useful function only if the function definition is conditional. If there is no conditional, the least fixed point will be the function $\bot$. In languages that provide lazy evaluation, the conditional is not always required.

3.6.5 Fixed Point Operators

We can avoid having to write out an equation and then specify a solution of it by defining a fixed point operator, as follows.

Definition 28 (Fixed Point Operator) *If f is a continuous function, then $\mu x \,.\, f(x)$ is the smallest solution of the equation $x = f(x)$. μ is called a* **fixed point operator***.*

An alternative notation for $\mu x \,.\, e$ is "fix $(\lambda x \,.\, e)$".

For example, we can define the factorial function as

$$\textit{factorial} \quad = \quad \mu f \,.\, \lambda x \,.\, \texttt{if } x = 0 \texttt{ then } 1 \texttt{ else } x \times f(x-1)$$

Referring back to the denotational semantics of L_1 (Figure 9 on page 10), we can improve the definition of the `while` commmand:

$$\mathcal{C}\,[\![\texttt{while } b \texttt{ do } c]\!] \quad = \quad \mu X \,.\, \lambda\sigma \,.\, \mathcal{V}\,[\![b\,]\!]\,\sigma \to (X \circ \mathcal{C}\,[\![c\,]\!])\sigma,\ \sigma$$

We consider this definition in more detail in Section 4.

3.6.6 Non-computable Functions

Let $H : (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to \mathbb{T}$ be defined by $H(f, n) \triangleq f(n) = \perp$. If H was computable, we could use it to solve the "halting problem". But H is not computable because it is non-monotonic. To see this, define:

$$\begin{aligned} f(n) &= \perp \\ g(n) &= \begin{cases} \perp, & \text{if } n = \perp; \\ n, & \text{otherwise.} \end{cases} \end{aligned}$$

(That is, g is the identity function.) Then $f \sqsubseteq g$ but, if $n \neq \perp$, we have

$$\begin{aligned} H(f, n) &= \mathsf{T} \\ H(g, n) &= \mathsf{F} \end{aligned}$$

so that H is non-monotonic.

3.6.7 Strict Functions

Definition 29 *A function f is* **strict** *if $f(\perp) = \perp$.*

The `if` command in most programming languages is an example of a non-strict function. We expect, for example:

$$\begin{aligned} \texttt{if true then } c_1 \texttt{ else } \perp &= c_1 \\ \texttt{if false then } \perp \texttt{ else } c_2 &= c_2 \end{aligned}$$

We exploit the non-strictness of `if` commands in programs such as

$$\texttt{if } x = 0 \texttt{ then error else } y = 1/x.$$

Some programming languages provide non-strict versions of `and` and `or`. For example, in C: 0 `&&` $\perp$ yields 0 and 1 `||` $\perp$ yields 1 (where 0 denotes F and 1 denotes T, as in C, and $\perp$ stands for an expression whose evaluation might fail).

Exercise 25 Consider the function

$$f(x, y) = \texttt{if } x \neq \perp \texttt{ then } x \texttt{ else if } y \neq \perp \texttt{ then } y \texttt{ else } \perp.$$

Is f computable? □

Exercise 26 Consider the function

$$g(x, y) = \begin{cases} x, & \text{if } x \neq \perp \text{ and } y = \perp, \\ y, & \text{if } x = \perp \text{ and } y \neq \perp, \\ x, & \text{otherwise.} \end{cases}$$

Is g computable? □

3.6.8 Defining Continuous Functions on a Domain

It would be very inconvenient if we had to prove that every function we introduce is continuous. Fortunately, most of the functions that can be easily defined are continuous. Here, we define constructions that yield a wide variety of functions that are continuous. Strictly speaking, of course, we should prove that each construction yields a continuous function. For further discussion, see Winskel (1993, pages 136–139).

Definition 30 (Continuity of Expressions) *An expression e is* **continuous in the variable** $x \in D$ *if the function $\lambda x \,.\, e : D \to E$ is continuous.*

An expression e is **continuous in its variables** *if it is continuous in all of its free variables.*

Functions constructed according to the following rules are continuous.

- ▷ A constant expression is continuous.
- ▷ An expression consisting of a single variable is continuous. Let the expression by the variable y and consider $f = \lambda x \,.\, y$. If $x = y$, f is the identity function, which is continuous. If $x \neq y$, then f is a constant function, which is also continuous.
- ▷ If f is a function constant and e is an expression continuous in its variables, then the application $f(e)$ is continuous in its variables.
- ▷ If e is continuous in its variables, then the function $\lambda x \,.\, e$ is continuous in its variables.
- ▷ If the expressions $e_1 \in D_1, \ldots, e_n \in D_n$ are continuous in their variables, then the tuple $(e_1, \ldots, e_n) \in D_1 \times \cdots \times D_n$ is continuous in its variables.
- ▷ If the expressions x, y, and z are continuous in their variables, the conditional expression $x \to y,\ z$ is continuous in its variables.

4 Denotational Semantics Revisited

We can use the mathematical machinery of the previous section to provide a useful denotational semantics of L_1. We begin with some notational conventions.

4.1 High Order Functions

Consider $4 + 5$ where $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. We could write this expression as $+(4, 5)$, or $+\ 4\ 5$ or $(+4)5$.

We can abstract this expression, making 5 a variable: $\lambda x\,.\,(+4)x$. But, by the η rule, $\lambda x\,.\,Mx = M$ (provided that x is not free in M). Thus $(+4)$ is a function that adds 4 to its argument. We have:

$$\begin{aligned} + &: \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \\ +4 &: \mathbb{N} \to \mathbb{N} \\ +\,4\,5 &: \mathbb{N} \end{aligned}$$

In general, for any function $f : A \times B \to C$, there is a corresponding function $g : A \to (B \to C)$ such that, for all $x \in A$ and $y \in B$,

$$f(x, y) \quad = \quad g(x)(y) \in C$$

We say that g is the **curried** form of f (after the American logician, Haskell Curry).

Curried functions have similar advantages to λ notation. Let $\mathcal{C}[\![\ ]\!] : \texttt{Com} \times \Sigma \to \Sigma$ be a semantic function, so that $\mathcal{C}(c, \sigma) = \sigma'$. This gives only the meaning of a progrqm with respect to a state. What we really want is the meaning of the program. In other words, we prefer $\mathcal{C}[\![\ ]\!] : \texttt{Com} \to (\Sigma \to \Sigma)$. Instead of writing

$$\mathcal{C}[\![c]\!]\sigma \quad = \quad \sigma'$$

we write

$$\mathcal{C}[\![c]\!] \quad = \quad \lambda\sigma\,.\,\sigma'$$

and we try to reason with $\mathcal{C}[\![c\ ]\!]$.

Notational Conventions We assume that $A \to B \to C$ means $A \to (B \to C)$ and that fxy means $(fx)y$. These notations are consistent:

$$\begin{aligned} fxy &: C \\ fx &: B \to C \\ f &: A \to B \to C \end{aligned}$$

A λ abstraction with two λs denotes a function with two arguments. For example, $\lambda x\,.\,\lambda y\,.\,M$ can take two arguments: the first argument corresponds to x and the second to y.

$$\begin{aligned} & (\lambda x\,.\,\lambda y\,.\ \ldots x \ldots y \ldots)ab \\ = \quad & ((\lambda \underline{x}\,.\,\lambda y\,.\ \ldots \underline{x} \ldots y \ldots)\underline{a})b \\ = \quad & (\lambda \underline{y}\,.\ \ldots a \ldots \underline{y} \ldots)\underline{b} \\ = \quad & \ldots a \ldots b \ldots \end{aligned}$$

We can abbreviate $\lambda x\,.\,\lambda y\,.\,M$ to $\lambda xy\,.\,M$.

4.2 Syntactic Domains

As before, we have a domain `Loc`, with x as a typical member, of locations.

We merge numeric and boolean expressions into a single syntactic class called `Exp` defined by

$$e \quad = \quad \texttt{true} \mid \texttt{false} \mid n \mid x \mid e + e \mid e \leq e$$

where, as before, $n \in \texttt{Num}$ and $x \in \texttt{Loc}$.

The syntactic class of commands, `Com`, is defined by

$$c \quad = \quad \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c$$

Note that there is no longer a syntactic constraint that the expressions in a `if` or `while` statement should be boolean. The semantics will ensure that the program will fail if they are not boolean.

4.3 Semantic Domains

Values of expressions are members of the domain

$$\mathbb{E}_\perp \quad = \quad \mathbb{N}_\perp + \mathbb{T}_\perp$$

This is a separated sum (see Figure 19) containg $\perp_\mathbb{N}$ (the unknown natural number), $\perp_\mathbb{T}$ (the unknown truth value), and $\perp_\mathbb{E}$ (the bottom of the sum domain).

As before, Σ, with typical member σ is the set of states. A state σ is a function from locations to values: $\sigma : \texttt{Loc} \to \mathbb{E}_\perp$. A state can be empty ($\langle\rangle$), or be defined by extension ($\sigma \oplus \langle x = \xi \rangle$). Attempting to apply a state outside its domain (in other words, using an uninitialized variable) leads to failure:

$$\begin{aligned} \langle\rangle &= \lambda y \,.\, \perp_\mathbb{E} \\ \sigma \oplus \langle x = \xi \rangle &= \lambda y \,.\, x = y \to \xi, \sigma(y) \end{aligned}$$

4.4 Semantic Equations

Before giving the semantics equations, we define the conditional expression $x \to y, z$ so that it yields $\perp_\mathbb{E}$ if $x \notin \mathbb{T}$, as shown in the following table. The value of the function does not depend on the value of an entry shown as "–".

x	y	z	$x \to y, z$
T	Y	–	Y
F	–	Z	Z
other	–	–	$\perp$

Figure 22 shows the new semantic equations. They are quite similar to the equations of Figure 9 on page 10, but there are some important differences.

The semantic functions are defined in such a way that the meaning of a program is a function:

$$\begin{aligned} &\text{Expressions:} \quad \mathcal{V}[\![\cdot]\!] \;:\; \texttt{Exp} \to \Sigma \to \mathbb{E}_\perp \\ &\text{Commands:} \quad \mathcal{C}[\![\cdot]\!] \;:\; \texttt{Com} \to \Sigma \to \Sigma \end{aligned}$$

$$\begin{aligned}
\mathcal{V}[\![n]\!] &= \lambda\sigma\,.\,\mathrm{val}\,n && (79)\\
\mathcal{V}[\![\texttt{true}]\!] &= \lambda\sigma\,.\,\mathsf{T} && (80)\\
\mathcal{V}[\![\texttt{false}]\!] &= \lambda\sigma\,.\,\mathsf{F} && (81)\\
\mathcal{V}[\![x]\!] &= \lambda\sigma\,.\,\sigma(x) && (82)\\
\mathcal{V}[\![e_1 + e_2]\!] &= \lambda\sigma\,.\,\mathcal{V}[\![e_1]\!]\sigma + \mathcal{V}[\![e_2]\!]\sigma && (83)\\
\mathcal{V}[\![e_1 \le e_2]\!] &= \lambda\sigma\,.\,\mathcal{V}[\![e_1]\!]\sigma \le \mathcal{V}[\![e_2]\!]\sigma && (84)\\
\mathcal{C}[\![\texttt{skip}]\!] &= \lambda\sigma\,.\,\sigma && (85)\\
\mathcal{C}[\![x := e]\!] &= \lambda\sigma\,.\,\sigma \oplus \langle x = \mathcal{V}[\![e]\!]\sigma \rangle && (86)\\
\mathcal{C}[\![c_1 ; c_2]\!] &= \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] && (87)\\
\mathcal{C}[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] &= \lambda\sigma\,.\,\mathcal{V}[\![e]\!]\sigma \rightarrow \mathcal{C}[\![c_1]\!]\sigma,\ \ \mathcal{C}[\![c_2]\!]\sigma && (88)\\
\mathcal{C}[\![\texttt{while}\ e\ \texttt{do}\ c]\!] &= \mu X\,.\,\lambda\sigma\,.\,\mathcal{V}[\![e]\!]\sigma \rightarrow (X \circ \mathcal{C}[\![c]\!])\sigma,\ \sigma && (89)
\end{aligned}$$

Figure 22: Revised denotational semantics for L_1

- ▷ The operators $+ : \mathbb{E}_\perp \times \mathbb{E}_\perp \rightarrow \mathbb{E}_\perp$ and $\le : \mathbb{E}_\perp \times \mathbb{E}_\perp \rightarrow \mathbb{E}_\perp$ in the right sides of (83) and (84) are defined as follows:

$$\begin{aligned}
x + y &= x \in \mathbb{N}_\perp \wedge y \in \mathbb{N}_\perp \rightarrow x + y, \perp_{\mathbb{E}}\\
x \le y &= x \in \mathbb{N}_\perp \wedge y \in \mathbb{N}_\perp \rightarrow x \le y, \perp_{\mathbb{E}}
\end{aligned}$$

  Thus $+$ and $\le$ are *strict in both arguments*. These definitions are simplified. Formally, elements of $\mathbb{E}_\perp$ carry tags (see Section 3.6.1) and we should use projections to obtain their values. Details of this kind become important when we construct software to manipulate semantic equations.
- ▷ The rule for assignment in Figure 9 was incorrect: in the corrected version, (86), the right side of the assignment (e) is evaluated before being assigned to the left side (x).
- ▷ In (87), we use functional composition to give the meaning of a sequence without explicit reference to the state.
- ▷ The meaning of the `if` command, defined by (88), uses the new version of the conditional function. Consequently, it fails if the condition e is not Boolean.
- ▷ The `while` command, defined by (89), is defined with the fixed point operator. The definition is compositional. The `while` command also fails if the conditional expression does not yield a Boolean value.

Since the meaning of a program is a function, we have a natural concept of program equivalence: programs are equivalent if they are equal as functions. Functions are equal if they give the same value everywhere in their domains. We use metabrackets on the left side of the following definition simply to indicate program text.

$$[\![c_1]\!] = [\![c_2]\!] \iff \forall\sigma \in \Sigma\,.\,\mathcal{C}[\![c_1]\!]\sigma = \mathcal{C}[\![c_2]\!]\sigma.$$

4.5 Using the Semantics

We can use the semantic equations to prove general properties of the programs. The properties derived in this section are trivial and uninteresting, but then so is the language.

If evaluation of a condition does not yield a boolean result, the program fails. For example:

$$\begin{aligned}\mathcal{C}[\![\texttt{if 3 then } c_1 \texttt{ else } c_2]\!] &= \lambda\sigma\,.\,\mathcal{V}[\![3]\!]\sigma \rightarrow \mathcal{C}[\![c_1]\!]\sigma,\ \mathcal{C}[\![c_2]\!]\sigma \\ &= \lambda\sigma\,.\,3 \rightarrow \mathcal{C}[\![c_1]\!]\sigma,\ \mathcal{C}[\![c_2]\!]\sigma \\ &= \lambda\sigma\,.\,\bot_\Sigma\end{aligned}$$

using the new definition of the conditional function.

Lemma 31 *For any command, c:*

$$[\![\texttt{skip}; c]\!] = [\![c; \texttt{skip}]\!] = [\![c]\!].$$

Proof For all states, σ, since $\mathcal{C}[\![\texttt{skip}]\!]$ is the identity function:

$$\begin{aligned}\mathcal{C}[\![\texttt{skip}; c]\!]\sigma &= (\mathcal{C}[\![c]\!] \circ \mathcal{C}[\![\texttt{skip}]\!])\sigma \\ &= \mathcal{C}[\![c]\!](\mathcal{C}[\![\texttt{skip}]\!]\sigma) \\ &= \mathcal{C}[\![c]\!]\sigma\end{aligned}$$

and

$$\begin{aligned}\mathcal{C}[\![c; \texttt{skip}]\!]\sigma &= (\mathcal{C}[\![\texttt{skip}]\!] \circ \mathcal{C}[\![c]\!])\sigma \\ &= \mathcal{C}[\![\texttt{skip}]\!](\mathcal{C}[\![c]\!]\sigma) \\ &= \mathcal{C}[\![c]\!]\sigma\end{aligned}$$

□

Lemma 32 *The program* `while true do skip`*, executed in any state, yields the undefined state.*

Proof From the semantic equations:

$$\mathcal{C}[\![\texttt{while true do skip}]\!] = \mu X\,.\,\lambda\sigma\,.\,\mathcal{V}[\![\texttt{true}]\!]\sigma \rightarrow (X \circ \mathcal{C}[\![\texttt{skip}]\!])\sigma,\ \sigma$$

Since

$$\begin{aligned}\mathcal{V}[\![\texttt{true}]\!]\sigma &= \mathsf{T} \\ \mathsf{T} \rightarrow x, y &= x \\ \text{and}\quad X \circ \mathcal{C}[\![\texttt{skip}]\!] &= X\end{aligned}$$

we can simplify the right hand side so that

$$\begin{aligned}\mathcal{C}[\![\texttt{while true do skip}]\!] &= \mu X\,.\,\lambda\sigma\,.\,X\sigma \\ &= \mu X\,.\,X.\end{aligned}$$

The desired value of X is the smallest solution of $f(X) = X$ which, by Tarski's theorem is $f^\infty(\bot) = f(\bot) = \bot$. Thus

$$\mathcal{C}[\![\texttt{while true do skip}]\!] = \lambda\sigma\,.\,\bot.$$

□

We can also establish the more general result that connects the meanings of the `while` and `if` commands.

Lemma 33 *The recursive definition of the* `while` *statement is consistent with the denotational semantics:*

$$[\![\,\texttt{while}\ e\ \texttt{do}\ c\,]\!] \quad = \quad [\![\,\texttt{if}\ e\ \texttt{then}\ (c;\texttt{while}\ e\ \texttt{do}\ c)\ \texttt{else}\ \texttt{skip}\,]\!].$$

Proof Let

$$W(X) \quad = \quad \lambda\sigma\,.\,\mathcal{V}\,[\![\,e\,]\!]\,\sigma \to (X \circ \mathcal{C}\,[\![\,c\,]\!]\,)\sigma,\ \sigma$$

so that $\mathcal{C}\,[\![\,\texttt{while}\ e\ \texttt{do}\ c\,]\!]$ is the least fixed point of W. Then, using once again the fact that $\mathcal{C}\,[\![\,\texttt{skip}\,]\!]$ is the identity function:

$$\begin{array}{rcl}
\mathcal{C}\,[\![\,\texttt{while}\ e\ \texttt{do}\ c\,]\!] & = & W(\mathcal{C}\,[\![\,\texttt{while}\ e\ \texttt{do}\ c\,]\!]\,) \\
& = & \lambda\sigma\,.\,\mathcal{V}\,[\![\,e\,]\!]\,\sigma \to (\mathcal{C}\,[\![\,\texttt{while}\ e\ \texttt{do}\ c\,]\!] \circ \mathcal{C}\,[\![\,c\,]\!]\,)\sigma,\ \sigma \\
& = & \lambda\sigma\,.\,\mathcal{V}\,[\![\,e\,]\!]\,\sigma \to \mathcal{C}\,[\![\,c;\texttt{while}\ e\ \texttt{do}\ c\,]\!]\sigma,\ \ \mathcal{C}\,[\![\,\texttt{skip}\,]\!]\sigma \\
& = & \mathcal{C}\,[\![\,\texttt{if}\ e\ \texttt{then}\ (c;\texttt{while}\ e\ \texttt{do}\ c)\ \texttt{else}\ \texttt{skip}\,]\!].
\end{array}$$

□

Exercise 27 Use the semantic equations to show that the following pairs of programs are equivalent.

$$\begin{array}{rcl}
[\![\,\texttt{if}\ e\ \texttt{then}\ (c_1;c)\ \texttt{else}\ (c_2;c)\,]\!] & = & [\![\,(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2);c\,]\!] \\
[\![\,\texttt{while}\ e\ \texttt{do}\ (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\,]\!] & = & [\![\,\texttt{while}\ e\ \texttt{do}\ c_1\,]\!]
\end{array}$$

□

Exercise 28 Use the semantic equations to determine the conditions under which the following pair of programs are equivalent. The expressions e_1 and e_2 may contain the variables x_1 and x_2.

$$[\![\,x_1 := e_1; x_2 := e_2\,]\!] \quad = \quad [\![\,x_2 := e_2; x_1 := e_1\,]\!]$$

□

5 Language Extensions

In this section, we consider various extensions of the language L_1. The extended language is called L_2 and the main result of this section is a denotational semantics for L_2. This section also introduces a few new pieces of notation and domain theory.

5.1 Let Notation

We write

$$\texttt{let}\ x = e\ \texttt{in}\ y$$

to denote the expression "y with all occurrences of x replaced by e". Clearly, this expression is equivalent to both $y[e/x]$ and (by the β-rule) to $(\lambda x\,.\,y)e$. We could use any of the three notations, but the `let` version is usually the most readable.

The scope of the definition of x is the whole of y. Consequently, we do not need the parentheses in expressions such as

$$\texttt{let}\ x_1 = e_1\ \texttt{in}\ (\texttt{let}\ x_2 = e_2\ \texttt{in}\ y).$$

To restrict the scope of x, we must enclose the `let` expression in parentheses. In this example, the final x occurs free:

$$(\texttt{let}\ x = 3\ \texttt{in}\ x + x) \times x \quad = \quad 6 \times x.$$

We also use an extended version of the `let` notation in which several variables are bound at the same time. Suppose that $a = (a_1, \ldots, a_n)$ is a tuple. Then we write

$$\texttt{let}\ (x_1, \ldots, x_n) = a\ \texttt{in}\ E$$

as an abbreviation for

$$\texttt{let}\ x_1 = a_1\ \texttt{in}\ \texttt{let}\ x_2 = a_2 \quad \texttt{.....}\ \texttt{in}\ E$$

We can write the expression `let` $x = y$ `in` z in the alternative form

$$z\ \texttt{where}\ x = y.$$

This has the same meaning, but we will use it less because the scope of x is not as clear when `where` expressions are embedded in larger expressions.

5.2 Streams

We use streams as an abstract structure for modelling input and output, discussed in the next section.

If D is a domain, we define the domain D^* (streams of Ds) as

$$D^* \quad = \quad D^0 + D^1 + D^2 + \cdots$$

where

$$\begin{array}{rcll} D^0 &=& \{\,()\,\} & \text{(the empty stream)} \\ D^1 &=& \{\,(d_1) \mid d_1 \in D\,\} & \\ D^2 &=& \{\,(d_1, d_2) \mid d_1, d_2 \in D\,\} & \\ && \cdots\cdots & \\ D^n &=& \{\,(d_1, \ldots, d_n) \mid d_1, \ldots, d_n \in D\,\} & \end{array}$$

We define the following operations on D^*. Let $d = (d_1, \ldots, d_n)$.

$$\begin{array}{rcll} \mathrm{hd}\, d &=& d_1 \in D & \text{(Head of } d\text{.)} \\ \mathrm{tl}\, d &=& (d_2, \ldots, d_n) \in D^* & \text{(Tail of } d\text{.)} \\ \mathrm{null}\, d &=& \begin{cases} \mathsf{T}, & \text{if } d \text{ is the empty stream,} \\ \mathsf{F}, & \text{otherwise.} \end{cases} & \text{(empty test)} \\ d \cdot x &=& (d_1, \ldots, d_n, x) & \text{(affix } x \in D\text{)} \end{array}$$

5.3 Input and Output

Suppose that we add `read` and `write` commands to L_1 with the following informal meanings.

`read` x	Obtain a value in $\mathbb{E}$ from the input medium and store it at location x.
`write` e	Copy the value of the expression e to the output medium.

We represent a "medium" as an element of $\mathbb{E}^*$ and we extend the state to include an input stream and an output stream.

$$\begin{array}{rcl} \Sigma &=& \mathtt{Mem} \times \mathtt{Inp} \times \mathtt{Out} \\ \mathtt{Mem} &=& \mathtt{Loc} \to \mathbb{E}_\perp \\ \mathtt{Inp} &=& \mathbb{E}^* \\ \mathtt{Out} &=& \mathbb{E}^* \end{array}$$

A state, σ, is now a triplet consisting of a memory map (this was the entire state before), an input stream, and an output stream. We will write $\sigma = (m, i, o)$.

The operations that we have previously applied to states must now be applied to the memory component of the state only. The operations $m(x)$ and $m \oplus \langle x = e \rangle$ are defined in the same way as the corresponding operations on σ.

The semantic equations for input and output commands are:

$$\begin{array}{rcl} \mathcal{C}\,[\![\mathtt{read}\ x]\!] &=& \lambda\sigma\,.\,\mathtt{let}\ (m,i,o) = \sigma\ \mathtt{in}\ \mathrm{null}\, i \to \perp, (m \oplus \langle x = \mathrm{hd}\, i \rangle, \mathrm{tl}\, i, o) \\ \mathcal{C}\,[\![\mathtt{write}\, e]\!] &=& \lambda\sigma\,.\,\mathtt{let}\quad (m,i,o) = \sigma \\ && \qquad\ \mathtt{in}\quad \mathtt{let}\ E = \mathcal{V}\,[\![\, e\,]\!]\sigma\ \ \mathtt{in}\ E = \perp \to \perp, (m, i, o \cdot E) \end{array}$$

The command `read` x fails if the input stream is empty, otherwise it stores the head of the input stream at location x. The new state consists of the updated memory, the rest of the input stream, and the unchanged output stream.

The command `write` e fails if the value of e is undefined — it would not make sense to have undefined values in a stream. Otherwise, the new state has the value of e affixed to the output stream.

The model that we have adopted for input and output is unsatisfactory in two respects.

- ▷ The semantics yields a complete state $\sigma = (m, i, o)$ as the meaning of the program. For a language with input and output, a more natural meaning would be a function from inputs to outputs.
- ▷ Since the output stream is part of the state, we could write semantic equations that made use of values previously written. The semantic equations that we have given do not do this, but only as a matter of convention.

Both of these problems can be eliminated by the use of continuations.

Exercise 29 Add the expression `eof` to L_2: the value of `eof` is `true` if the input stream is empty and `false` otherwise. □

Exercise 30 Add the Boolean function `not` to L_2 with the same meaning as `not` in Pascal. □

Exercise 31 Determine the final state yielded by the program

$$\texttt{while not eof do } (\texttt{read}\, x; \texttt{write}\, x)$$

when it is executed with initial state $(\langle\rangle, (1, 2), ())$. [Hint: use Lemma 33.] □

Reading (Schmidt 1986, pages 88–90).

5.4 Value Domains

The state function that we have used so far is $\sigma : \texttt{Loc} \to \mathbb{E}_\perp$ and its range is $\mathbb{E}_\perp$, the values that we can store in a memory. This is adequate for L_1, in which the set of values that we can name is precisely the set of values that we can store, but it is inadequate for practical languages such as Pascal and C. In Pascal, for example, we can give names to constants, types, and variables, but we can store only variables. To make the required distinctions, we introduce several sets of values; the precise content of these sets will depend on the language that we are describing.

Definition 34 (Basic Values) *The set* `Bv` *of* **basic values** *consists of all the unstructured values that are used to define the language.*

Definition 35 (Denotable Values) *The set* `Dv` *of* **denotable values** *consists of values that can be given a name in the language.*

Definition 36 (Expressible Values) *The set* `Ev` *of* **expressible values** *consists of values that can be obtained by evaluating an expression other than a name.*

Definition 37 (Storable Values) *The set* `Sv` *of* **storable values** *consists of values that can be stored in and retrieved from memory and are accessible by programs as such.*

Examples The basic values that we have been using so far are truth values and natural numbers. Thus $\mathtt{Bv} = \mathbb{T}_\perp + \mathbb{N}_\perp$. Practical languages typically have integers, characters, and floating point numbers as basic values. There are a few languages in which strings are basic values, but in most languages strings are defined as arrays or sequences of characters.

In Pascal, constants are denotable (by `const` declarations) and expressible ('3' and '*true*' are constant expressions) but not storable. Types are similar. Variables are denotable (by `var` declarations), expressible (expressions such as $a[i]$ and $a\uparrow$ yield variables in appropriate contexts), and storable. Arrays are denotable, storable, but not expressible.

In C, constants are expressible but not denotable and not storable. (The `#define` mechanism provides a way of naming constants, but it is hardly practical to give a semantics of C that includes preprocessing.) Locations are denotable, expressible, and storable: all three are illustrated by the statement `p = &i`.

Exercise 32 Describe and compare the sets of basic values, denotable values, expressible values, and storable values for **two** languages. The languages should be different enough to make the comparison interesting: for example, you could compare C and Pascal, or Pascal and Turbo-Pascal.

If you choose C, do not include preprocessing. (The preprocessor makes everything denotable, including meaningless code fragments.) Arrays, pointers, and functions need careful consideration.

If you choose Turbo-Pascal, you may need a reference manual to check arcane features such as the @ operator. □

5.5 Declarations and Blocks

The model that we have been using for variables and memory is unsatisfactory. Since we cannot give a meaning to declarations, we cannot distinguish local and global variables. If we wanted to add procedures and functions, we would be unable to describe parameters and local variables.

L_1 does not require variables to be declared. It is difficult to check, without executing a program (or simulating its execution) whether variables are used correctly. To avoid this problem, many programming languages require that variables be declared before they are used.

We introduce a new syntactic class `Dec` of declarations with typical member d:

$$d \quad = \quad \epsilon \mid \mathtt{var}\, x \mid d; d$$

We will need a new semantic function

$$\mathcal{D}[\![\,\cdot\,]\!] \quad : \quad \mathtt{Dec} \rightarrow \mathtt{Env} \rightarrow \mathtt{Env}$$

to give a meaning to declarations.

In order to integrate declarations into programs (which we have defined as commands), we introduce a new kind of command called a *block* with syntax

$$c \quad = \quad \{\, d; c \,\} \mid \cdots$$

The block is similar to a C block, both syntactically and semantically. The scope of the declarations d is the command c; after the closing brace, the variables introduced in d are no longer accessible. A complete program is often a block, although we do not require this.

5.6 Environments

The mechanism that we introduce to give a meaning to declarations is the *environment.* We do this by separating the concepts of identifiers and locations. An environment is a mapping from identifiers to locations; a state is still a mapping from locations to values. Thus, a name may refer to different locations at different points in the program, and a location may be associated with more than one name.

We refine the model in the following way.

- ▷ There is a new syntactic domain of identifiers: `Ide`. Thus we now have $x \in$ `Ide` rather than $x \in$ `Loc`.
- ▷ There is a mapping from identifiers to locations, $\rho :$ `Ide` $\rightarrow$ `Loc`, called an *environment.*
- ▷ The state is a mapping from locations to storable values, $\sigma :$ `Loc` $\rightarrow$ `Sv`.

We can view the new situation like this:

$$\text{identifier} \xrightarrow{\rho} \text{location} \xrightarrow{\sigma} \text{value}$$

Environments enable us to model an important property of many programming languages: the location associated with a name may depend on where the name occurs in the program. It also allows us to express the fact that there may not be a location corresponding to an identifier.

If ρ is an environment and x is an identifier, then $\rho [\![x]\!]$ is the location assigned for x. To avoid the details of updating environments, we will informally define the environment $\rho \oplus [\![y]\!]$ to be the environment ρ with an additional binding for the variable y.

Note 38 It is not hard to provide a precise definition for the environment function. For example, we could model it as a pair (g, n) is which g is a set of identifier/address pairs and n is the next free address. The empty environment is $(\varnothing, 0)$ and $(\{ (x,0),(y,1) \}, 2)$ is an environment in which x has been assigned address 0, y has been assigned address 1, and the next free address is 2. Then

$$\rho \oplus [\![y]\!] \quad = \quad \texttt{let}\ (g,n) = \rho\ \texttt{in}\ (g \cup \{ (y,n) \}, n+1).$$

This definition is not quite good enough, because it allows an identifier to be introduced into an environment more than once. A complete definition would allow multiple scopes and would not allow duplicate definitions within a scope.

Reading (Schmidt 1986, pages 137–9), (Stoy 1977, pages 225–33).

5.7 A Semantics with Environments

For a language with declarations, the semantics must have environments. Semantic functions have an environment as an additional argument.

To evaluate an expression, we need an environment and a state; the result is an expressible value:

A typical semantic equation is written as

$$\begin{array}{lll} & \mathcal{F}[\![e]\!]\rho\sigma & = E, \\ \text{or} & \mathcal{F}[\![e]\!]\rho & = \lambda\sigma\,.\,E, \\ \text{or} & \mathcal{F}[\![e]\!] & = \lambda\rho\,.\,\lambda\sigma\,.\,E. \end{array}$$

where $\mathcal{F}$ stands for a typical semantic function. The intermediate form, $\mathcal{F}[\![e]\!]\rho$ corresponds roughly to a compiled form of the program. One of the tasks of a compiler is to choose locations (absolute or relative) for variables, so that the effect of compiled code depends only on the state — that is, the values in memory when the program is executed.

There are various ways of introducing declarations into the programming language. We could introduce a separate syntactic domain for declarations, giving a Pascal-like language in which declarations are strictly separated from commands. The approach that we will actually follow is to introduce declarations as a kind of command, which is closer to C style.

The new language is called L_2.

5.7.1 Syntactic Domains of L_2

The syntactic domains, with typical elements, are:

n	$\in$	`Num`	Numerals
x	$\in$	`Ide`	Identifiers
d	$\in$	`Dec`	Declarations
e	$\in$	`Exp`	Expressions
c	$\in$	`Com`	Commands

The syntactic domains are defined as follows.

$$\begin{array}{lll} d & = & \epsilon \mid \texttt{var}\,x \mid d;d \\ e & = & \texttt{true} \mid \texttt{false} \mid n \mid x \mid e+e \mid e \le e \\ c & = & \{\,d;c\,\} \mid \texttt{skip} \mid x := e \mid c;c \mid \texttt{if}\;e\;\texttt{then}\;c\;\texttt{else}\;c \mid \\ & & \texttt{while}\;e\;\texttt{do}\;c \mid \texttt{read}\,x \mid \texttt{write}\,e \end{array}$$

5.7.2 Semantic Domains of L_2

A state Σ is a triple consisting of: a map from locations to storable values; the input; and the output.

$\mathbb{T}_\perp$	$=$	$\{\,\perp, \mathsf{T}, \mathsf{F}\,\}$	Truth values with bottom
$\mathbb{N}_\perp$	$=$	$\{\,\perp, 0, 1, 2, \dots\,\}$	Natural numbers with bottom
`Bv`	$=$	$\mathbb{T}_\perp + \mathbb{N}_\perp$	Basic values
`Dv`	$=$	`Loc`	Denotable values are memory locations
`Ev`	$=$	`Bv`	Expressible values
`Sv`	$=$	`Bv`	Storable values
`Env`	$=$	`Ide` $\to$ `Loc`	Environments map identifiers to locations
Σ	$=$	(`Loc` $\to$ `Sv`) $\times$ `Inp` $\times$ `Out`	

5.7.3 Semantic Functions of L_2

Declarations update environments. Expressions are evaluated in the context of an environment and a state and yield an expressible value. Commands are evaluated in the context of an environment and a state and yield a new state.

Declarations:	$\mathcal{D}[\![\cdot]\!]$	:	$\texttt{Dec} \to \texttt{Env} \to \texttt{Env}$
Expressions:	$\mathcal{V}[\![\cdot]\!]$	:	$\texttt{Exp} \to \texttt{Env} \to \Sigma \to \texttt{Ev}$
Commands:	$\mathcal{C}[\![\cdot]\!]$	:	$\texttt{Com} \to \texttt{Env} \to \Sigma \to \Sigma$

5.7.4 Semantic Equations of L_2

Figure 23 shows the semantic equations for L_2. Note:

- ▷ (96), (101), (105): it is an error to refer to a variable that has not been declared.
- ▷ (99): a block is executed by first updating the environment with the new declarations and then executing the commands within the new environment.
- ▷ (101): an assignment alters the state but not the environment.

  An implementor could deduce from (99) and (101) that the compiler could process declarations but would have to defer the execution of assignments until execution.
- ▷ (102): we cannot define a command sequence by functional composition because evaluation of the first command does not yield an environment for the second. The function we need is

$$\lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{C}[\![c_2]\!]\,\rho(\mathcal{C}[\![c_1]\!]\rho\sigma)$$

  which simplifies to the form shown.

  We use the same technique in the equation for `while` e `do` c.
- ▷ (105): attempting to read when the input tream is empty leads to failure.
- ▷ (106): attempting to write an undefined value to the output stream leads to failure.

Exercise 33 Write a program that "implements" the denotational semantics of L_2 (Figure 23). Here is an informal specification.

Input:	(1)	A program in L_2.
	(2)	A state description.
Action:	(1)	Parse the input program and construct an abstract syntax tree (AST) for the program.
	(1)	Apply the semantic function $\mathcal{C}$ to the AST and the input state.
Output:		The updated state.

□

$$
\begin{array}{rcll}
\mathcal{D}[\![\epsilon]\!] & = & \lambda\rho\,.\,\rho & (90)\\
\mathcal{D}[\![\mathtt{var}\,x]\!] & = & \lambda\rho\,.\,\rho\oplus[\![x]\!] & (91)\\
\mathcal{D}[\![d_1;d_2]\!] & = & \mathcal{D}[\![d_2]\!]\circ\mathcal{D}[\![d_1]\!] & (92)\\
\\
\mathcal{V}[\![\mathtt{true}]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{T} & (93)\\
\mathcal{V}[\![\mathtt{false}]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{F} & (94)\\
\mathcal{V}[\![n]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathrm{val}[\![n]\!] & (95)\\
\mathcal{V}[\![x]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\rho[\![x]\!]=\bot\to\bot,\mathtt{let}\,(m,i,o)=\sigma\,\mathtt{in}\,m(\rho[\![x]\!]) & (96)\\
\mathcal{V}[\![e_1+e_2]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{V}[\![e_1]\!]\rho\sigma+\mathcal{V}[\![e_2]\!]\rho\sigma & (97)\\
\mathcal{V}[\![e_1\le e_2]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{V}[\![e_1]\!]\rho\sigma\le\mathcal{V}[\![e_2]\!]\rho\sigma & (98)\\
\\
\mathcal{C}[\![\{\,d;c\,\}]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{C}[\![c]\!](\mathcal{D}[\![d]\!]\rho)\sigma & (99)\\
\mathcal{C}[\![\mathtt{skip}]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\sigma & (100)\\
\mathcal{C}[\![x:=e]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\rho[\![x]\!]=\bot\to\bot, & (101)\\
 & & \quad\mathtt{let}\,(m,i,o)=\sigma\,\mathtt{in}\,(m\oplus\langle\rho[\![x]\!]=\mathcal{V}[\![e]\!]\rho\sigma\rangle,i,o) & \\
\mathcal{C}[\![c_1;c_2]\!] & = & \lambda\rho\,.\,\mathcal{C}[\![c_2]\!]\rho\circ\mathcal{C}[\![c_1]\!]\rho & (102)\\
\mathcal{C}[\![\mathtt{if}\;e\;\mathtt{then}\;c_1\;\mathtt{else}\;c_2]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{V}[\![e]\!]\rho\sigma\to\mathcal{C}[\![c_1]\!]\rho\sigma,\ \mathcal{C}[\![c_2]\!]\rho\sigma & (103)\\
\mathcal{C}[\![\mathtt{while}\;e\;\mathtt{do}\;c]\!] & = & \mu X\,.\,\lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{V}[\![e]\!]\rho\sigma\to(X\rho\circ\mathcal{C}[\![c]\!]\rho)\sigma,\sigma & (104)\\
\mathcal{C}[\![\mathtt{read}\;x]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\rho[\![x]\!]=\bot\to\bot, & (105)\\
 & & \quad\mathtt{let}\quad(m,i,o)=\sigma & \\
 & & \quad\mathtt{in}\quad\mathrm{null}\,i\to\bot,(m\oplus\langle\rho[\![x]\!]=\mathrm{hd}\,i\rangle,\ \mathrm{tl}\,i,o) & \\
\mathcal{C}[\![\mathtt{write}\,e]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathtt{let}\quad E=\mathcal{V}[\![e]\!]\rho\sigma & (106)\\
 & & \qquad\qquad\mathtt{in}\quad E=\bot\to\bot,\quad\mathtt{let}\quad(m,i,o)=\sigma & \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\mathtt{in}\quad(m,i,o\cdot E) &
\end{array}
$$

Figure 23: Denotational semantics for L_2

5.8 Constant Declarations

Adding constant declarations of the form $\mathtt{const}\,x = e$ requires a change to the semantic domains. Specifically, we must add truth values and natural numbers to Dv, the domain of denotable values:

$$\mathtt{Dv} \;=\; \mathtt{Loc}+\mathbb{T}_\bot+\mathbb{N}_\bot.$$

This affects the structure of environments, because the function $\rho : \mathtt{Ide} \to \mathtt{Loc}$ must be able to return a location, a truth value, or a natural number. The semantic equations do not change, but the meaning of $\rho[\![x]\!]$ is broadened.

Exercise 34 Work out the details, and design suitable notation for, updating an environment with a new binding that binds an identifier to either a location or a constant value. □

5.9 Initialized Declarations

Declarations of the form $\texttt{var}\,x$ do not eliminate the problem of uninitialized variables. There are two ways in which we might deal with an initialized declaration with syntax

$$\texttt{var}\,x = e.$$

The first way is to split the declaration into two parts: $\texttt{var}\,x$ and $x := e$. Then we can put the first part into the declaration section and the second part into the command section of a block. This kind of "syntactic preprocessing" does not alter the semantics of the language.

Alternatively, we could treat $\texttt{var}\,x = e$ as a new command that alters both the environment and the state. This approach requires changing the type of declarations to

$$\mathcal{D}\,[\![\,\cdot\,]\!] \quad : \quad \texttt{Dec} \to (\texttt{Env} \times \Sigma) \to (\texttt{Env} \times \Sigma)$$

and changing the semantic equations of the language accordingly.

Exercise 35 Give the signatures of the semantic functions and new semantic equations for L_2 extended with initialized variable declarations of the form $\texttt{var}\,x = e$. □

5.10 Compilation and Execution

We introduced environments to model the role of declarations and variables in programs. The separation between environments and states corresponds to the separation between compilation and execution. Declarations affect the environment and can be processed at compile-time; commands affect the state and must be processed at run-time.

Definition 39 *The* **static semantics** *of a language determines that part of the meaning of a program that can be determined without executing the program.*

Definition 40 *The* **dynamic semantics** *of a language determines that part of the meaning of a program that can be determined only by executing it.*

Questions such as "Will the program terminate?" and "Will the program attempt to read past the end of the input stream?" can be determined only by running it. (In the general case, that is: obviously, there are specific instances in which we can answer these questions by static analysis.)

The obvious question that arises is: what can we determine statically? This section provides a partial answer. We consider the ways in which a program might "go wrong" and derive conditions under which we can say "the program works" without actually running it.

An examination of the semantic equations of Figure 23 suggests the following possible causes of failure.

- ▷ Use of an undeclared variable, leading to $\rho\,[\![x\,]\!] = \bot$.
- ▷ Use of a numeric expression where a Boolean expression is required, leading to failure of the conditional $\cdot \to \cdot, \cdot$.
- ▷ Calling $\texttt{read}\,x$ when the input stream is empty.

▷ A non-terminating `while` command.

The first two causes of failure can be predicted by static analysis. We can derive a function

$$\mathcal{W}[\![\cdot]\!] \quad : \quad (\texttt{Dec} + \texttt{Exp} + \texttt{Com}) \to \texttt{Env} \to \mathbb{T}$$

with the property defined by Lemma 41 below. In practice, we need two functions. In addition to $\mathcal{W}$, we use an auxiliary function

$$\mathcal{B}[\![\cdot]\!] \quad : \quad \texttt{Exp} \to \mathbb{T}$$

which returns T if its argument must be a Boolean expression. The two functions are shown in Figures 24 and 25.

$$\mathcal{B}[\![\texttt{true}]\!] = \mathsf{T} \tag{107}$$
$$\mathcal{B}[\![\texttt{false}]\!] = \mathsf{T} \tag{108}$$
$$\mathcal{B}[\![n]\!] = \mathsf{F} \tag{109}$$
$$\mathcal{B}[\![x]\!] = \mathsf{F} \tag{110}$$
$$\mathcal{B}[\![e_1 + e_2]\!] = \mathsf{F} \tag{111}$$
$$\mathcal{B}[\![e_1 \le e_2]\!] = \mathsf{T} \tag{112}$$

Figure 24: Definition of $\mathcal{B}$.

Lemma 41 *If $\mathcal{W}[\![c]\!] = \lambda\rho\,.\,\mathsf{T}$, the program c will not fail because a variable has not been declared or because an expression that is expected to return a Boolean result does not do so.*

Proof The proof is by structural induction over the syntax. The inductive hypothesis (IH) for a command c is that, if $\mathcal{W}[\![c]\!] = \lambda\rho\,.\,\mathsf{T}$, then c cannot fail for the reasons given.

Case ϵ. The empty declaration cannot cause failure.

Case d. A single declaration cannot cause failure.

Case $d; d$. Since a single declaration cannot cause failure, multiple declarations cannot cause failure.

Case `true`. Evaluation of a constant expression cannot cause failure.

Case `false`. Evaluation of a constant expression cannot cause failure.

Case n. Evaluation of a number cannot cause failure.

Case x. If $\mathcal{W}[\![x]\!] = \lambda\rho\,.\,\mathsf{T}$, then $x \in \operatorname{dom}\rho$. Consequently, the variable x has been declared and its evaluation cannot cause failure.

Case $e_1 + e_2$. If $\mathcal{W}[\![e_1 + e_2]\!] = \lambda\rho\,.\,\mathsf{T}$, then $\mathcal{W}[\![e_1]\!] = \lambda\rho\,.\,\mathsf{T}$ and $\mathcal{W}[\![e_2]\!] = \lambda\rho\,.\,\mathsf{T}$. By IH, neither of these expressions can cause failure. Consequently, $e_1 + e_2$ cannot cause failure. (Note that we do not consider the possibility of overflow.)

$$\begin{aligned}
\mathcal{W}[\![\epsilon]\!] &= \lambda\rho\,.\,\mathsf{T} && (113)\\
\mathcal{W}[\![d]\!] &= \lambda\rho\,.\,\mathsf{T} && (114)\\
\mathcal{W}[\![d;d]\!] &= \lambda\rho\,.\,\mathsf{T} && (115)\\
\mathcal{W}[\![\texttt{true}]\!] &= \lambda\rho\,.\,\mathsf{T} && (116)\\
\mathcal{W}[\![\texttt{false}]\!] &= \lambda\rho\,.\,\mathsf{T} && (117)\\
\mathcal{W}[\![n]\!] &= \lambda\rho\,.\,\mathsf{T} && (118)\\
\mathcal{W}[\![x]\!] &= \lambda\rho\,.\,x \in \operatorname{dom}\rho && (119)\\
\mathcal{W}[\![e_1 + e_2]\!] &= \lambda\rho\,.\,\mathcal{W}[\![e_1]\!]\,\rho \wedge \mathcal{W}[\![e_2]\!]\,\rho && (120)\\
\mathcal{W}[\![e_1 \leq e_2]\!] &= \lambda\rho\,.\,\mathcal{W}[\![e_1]\!]\,\rho \wedge \mathcal{W}[\![e_2]\!]\,\rho && (121)\\
\mathcal{W}[\![d;c]\!] &= \lambda\rho\,.\,\mathcal{W}[\![d]\!]\,\rho \wedge \mathcal{W}[\![c]\!]\,\rho && (122)\\
\mathcal{W}[\![\texttt{skip}]\!] &= \lambda\rho\,.\,\mathsf{T} && (123)\\
\mathcal{W}[\![x := e]\!] &= \lambda\rho\,.\,\mathcal{W}[\![x]\!]\,\rho \wedge \mathcal{W}[\![e]\!]\,\rho && (124)\\
\mathcal{W}[\![c_1;c_2]\!] &= \lambda\rho\,.\,\mathcal{W}[\![c_1]\!]\,\rho \wedge \mathcal{W}[\![c_2]\!]\,\rho && (125)\\
\mathcal{W}[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] &= \lambda\rho\,.\,\mathcal{B}[\![e]\!]\,\rho \wedge \mathcal{W}[\![c_1]\!]\,\rho \wedge \mathcal{W}[\![c_2]\!]\,\rho && (126)\\
\mathcal{W}[\![\texttt{while}\ e\ \texttt{do}\ c]\!] &= \lambda\rho\,.\,\mathcal{B}[\![e]\!]\,\rho \wedge \mathcal{W}[\![c]\!]\rho && (127)\\
\mathcal{W}[\![\texttt{read}\ x]\!] &= \mathcal{W}[\![x]\!] && (128)\\
\mathcal{W}[\![\texttt{write}\ e]\!] &= \mathcal{W}[\![e]\!] && (129)
\end{aligned}$$

Figure 25: Definition of $\mathcal{W}$.

Case $e_1 \leq e_2$. If $\mathcal{W}[\![e_1 \leq e_2]\!] = \lambda\rho\,.\,\mathsf{T}$, then $\mathcal{W}[\![e_1]\!] = \lambda\rho\,.\,\mathsf{T}$ and $\mathcal{W}[\![e_2]\!] = \lambda\rho\,.\,\mathsf{T}$. By IH, neither of these expressions can cause failure. Consequently, $e_1 \leq e_2$ cannot cause failure.

Case $d;c$. If $\mathcal{W}[\![d;\ c]\!] = \lambda\rho\,.\,\mathsf{T}$, then $\mathcal{W}[\![d]\!] = \lambda\rho\,.\,\mathsf{T}$ and $\mathcal{W}[\![c]\!] = \lambda\rho\,.\,\mathsf{T}$. By IH, neither of these components can cause failure, and so the block cannot cause failure.

Case `skip`. The `skip` command cannot cause failure.

Case $x := e$. If $\mathcal{W}[\![x := e]\!] = \lambda\rho\,.\,\mathsf{T}$, then $\mathcal{W}[\![x]\!] = \lambda\rho\,.\,\mathsf{T}$ and $\mathcal{W}[\![e]\!] = \lambda\rho\,.\,\mathsf{T}$. By IH, neither of these components can cause failure, and so the block cannot cause failure.

Case $c_1;c_2$. If $\mathcal{W}[\![c_1;c_2]\!] = \lambda\rho\,.\,\mathsf{T}$, then $\mathcal{W}[\![c_1]\!] = \lambda\rho\,.\,\mathsf{T}$ and $\mathcal{W}[\![c_2]\!] = \lambda\rho\,.\,\mathsf{T}$. By IH, neither of these components can cause failure, and so the block cannot cause failure.

Case `if` e `then` c_1 `else` c_2. If $\mathcal{W}[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] = \lambda\rho\,.\,\mathsf{T}$, then $\mathcal{B}[\![e]\!] = \lambda\rho\,.\,\mathsf{T}$. Thus e is a Boolean expression and its evaluation cannot cause failure. By IH applied to c_1 and c_2, the entire command cannot fail.

Case `while` e `do` c. If $\mathcal{W}[\![\texttt{while}\ e\ \texttt{do}\ c]\!] = \lambda\rho\,.\,\mathsf{T}$, then $\mathcal{B}[\![e]\!] = \mathsf{T}$. Thus e is a Boolean expression and its evaluation cannot cause failure. By IH applied to c, the entire command cannot fail.

Case `read` x. Since x cannot fail, the command cannot fail.

Case `write` e. Since e cannot fail, the command cannot fail.

□

The function $\mathcal{W}$, the lemma, and the proof are all rather trivial. The point of the example is to show that we are not restricted to writing semantic functions over the syntax: we have a useful mechanism that enables us to describe properties of a language succinctly and precisely.

The function $\mathcal{W}$ is *conservative*: as Lemma 41 claims, if $\mathcal{W}[\![c]\!] = \lambda\rho \,.\, \mathsf{T}$, we can be certain that the program will not fail for the reasons given. The converse of the lemma, however, is false: there are programs for which $\mathcal{W}[\![c]\!] = \lambda\rho \,.\, \mathsf{F}$ but which nevertheless run correctly. To obtain a more precise version of $\mathcal{W}$, we would have to include types in declarations.

The function $\mathcal{W}$ corresponds to the "static checking" performed by a compiler. Since the definition of $\mathcal{W}$ does not mention the state, we do not need to execute (or simulate the execution of) the program c to determine the value of $\mathcal{W}[\![c]\!]$. The trade-off, of course, is that we cannot use $\mathcal{W}$ to detect dynamic errors such as looping.

Exercise 36 Give an example of a program c for which $\mathcal{W}[\![c]\!] = \lambda\rho \,.\, \mathsf{F}$ although the program runs correctly. □

Reading (Schmidt 1986, pages 137–139).

6 More Complications

As we introduce more features of practical programming languages, the semantic domains and functions become increasingly complex. To reduce the complexity, we will introduce new language constructs in this section by means of language fragments, rather than complete — although simple — languages such as L_1 and L_2.

6.1 L and R Values

Consider the assignment $x := x + 1$. The meaning of this command, according to Figure 23 but ignoring input and output, is

$$\mathcal{C}\,[\![x := x+1]\!] \quad = \quad \lambda\rho\,.\,\lambda\sigma\,.\,\rho\,[\![x]\!] = \bot \to \bot,\ \sigma \oplus \langle \rho\,[\![x]\!] = \mathcal{V}\,[\![x+1]\!]\,\rho\sigma \rangle$$

Note the asymmetry: on the left side, we use $\rho\,[\![x]\!]$ to obtain the *address* (location) of x and, on the right side, we (implicitly) use $\mathcal{V}\,[\![x]\!]\,\rho\,\sigma$ to obtain the *value* of x.

It is useful to generalize the mechanisms for obtaining addresses and values. We need the generalized version for assignment statements in which the left side is not a simple variable (for example, $a[i] := \cdots$ and $r.f := \cdots$) and in other contexts. We do this by defining two new evaluation functions, $\mathcal{L}$ and $\mathcal{R}$, with the same type as $\mathcal{V}$. For a variable, x:

$$\begin{aligned}
\mathcal{L}\,[\![x]\!] &= \lambda\rho\sigma\,.\,\rho\,[\![x]\!] \\
\mathcal{R}\,[\![x]\!] &= \lambda\rho\sigma\,.\,\sigma(\rho\,[\![x]\!]) \\
&= \lambda\rho\sigma\,.\,\sigma(\mathcal{L}\,[\![x]\!]\,\rho\,\sigma)
\end{aligned}$$

We can use these functions to rewrite the semantics of the assignment statement:

$$\mathcal{C}\,[\![e_1 := e_2]\!] \quad = \quad \lambda\rho\sigma\,.\,\sigma \oplus \langle \mathcal{L}\,[\![e_1]\!]\,\rho\,\sigma = \mathcal{R}\,[\![e_2]\!]\,\rho\,\sigma \rangle$$

We have used e_1 rather than x on the left side to indicate that the left side is not necessarily a simple variable.

In semantics (as in C), we refer to the address (location) of an expression as an *lvalue* (pronounced “ell-value”) and the value of the expression as an *rvalue* (pronounced “ar-value”). It is always possible to convert an lvalue to an rvalue by *dereferencing*, but rvalues cannot in general be converted to lvalues.

If p is an lvalue, the corresponding rvalue is $\sigma\,p$ in the semantics, `*`p in C, and $p\uparrow$ in Pascal.

Exercise 37 Write down all possible classes of lvalues and rvalues for Pascal and C. □

Exercise 38 Give an example of a Pascal construct that has an lvalue but no rvalue. (Hint: when a procedure with a `var` parameter is called, the corresponding argument must be an lvalue.) □

6.2 Extending the Environment

The environment is a map from identifiers to denotable values, `Ide` → `Dv`. So far, we have assumed `Dv` = `Loc`, but we will need to extend `Dv` to include other denotable values such as constants, arrays, records, pointers, procedures, and functions. We will continue to write

$\rho \oplus \langle x = e \rangle$ to stand for the environment ρ with x (re)bound to the value e, whatever the type of e. We will use "recognizer" functions to check the result of applying an environment to an identifier. For example:

$$\mathcal{L}[\![x]\!] = \lambda\rho\sigma \,.\, \text{isloc}\; x \rightarrow \rho[\![x]\!], \perp.$$

So far we have defined the result of any failing operation to be $\perp$. We can provide more precision by introducing a constant *error* to indicate a static error in the program. For example, it would be better to write the equation above in the following form:

$$\mathcal{L}[\![x]\!] = \lambda\rho\sigma \,.\, \text{isloc}\; x \rightarrow \rho[\![x]\!], \mathit{error}.$$

6.2.1 Constants

We can introduce constants using Pascal-like syntax:

$$d = \texttt{const}\, x = e \mid \cdots$$

Constant declarations imply basic values in the domain of denotable values:

$$\texttt{Dv} = \texttt{Loc} + \texttt{Bv}$$

In the semantics, a constant declaration introduces a new binding into the environment as usual, a constant has no lvalue, and the rvalue of a constant is simply its value in the environment.

$$\begin{aligned}
\mathcal{D}[\![\texttt{const}\ x = e]\!] &= \lambda\rho \,.\, \rho \oplus \langle x = e \rangle \\
\mathcal{L}[\![x]\!] &= \lambda\rho\sigma \,.\, \text{isconst}\; x \rightarrow \mathit{error}, \ldots \\
\mathcal{R}[\![x]\!] &= \lambda\rho\sigma \,.\, \text{isconst}\; x \rightarrow \rho[\![x]\!], \ldots
\end{aligned}$$

Exercise 39 The semantics of a constant declaration given above assume that e is a denotable value. Show how to generalize the semantics so that e can be an expression. There are two cases: static expressions that can be evaluated by the compiler, and dynamic expressions, that cannot be evaluated until run-time. □

6.3 Arrays

The following declaration is suitable for arrays without bounds-checking.

$$\begin{aligned}
d &= \texttt{array}\, x \mid \cdots \\
e &= x[e] \mid \cdots
\end{aligned}$$

One possible semantic interpretation of an array is a function from natural numbers to locations.

$$\begin{aligned}
\texttt{Dv} &= \texttt{Array} + \cdots \\
\texttt{Array} &= \mathbb{N} \rightarrow \texttt{Loc}
\end{aligned}$$

The lvalue of $x[e]$ is obtained by evaluating e to obtain a natural number, and using this number as the argument of the array function. The rvalue is obtained, as usual, by applying the state function to the location $x[e]$.

$$\begin{aligned}
\mathcal{L}[\![x\ [e]]\!] &= \lambda\rho\sigma\ .\ \text{isarray}\ x \to \rho[\![x]\!](\mathcal{V}[\![e]\!]\ \rho\,\sigma), \mathit{error} \\
\mathcal{R}[\![x\ [e]]\!] &= \lambda\rho\sigma\ .\ \text{isarray}\ x \to \sigma(\rho[\![x]\!](\mathcal{V}[\![e]\!]\ \rho\,\sigma)), \mathit{error}
\end{aligned}$$

If we allow any basic value to be a subscript, as in Pascal, we would define arrays to have type $\texttt{Bv} \to \texttt{Loc}$. If we wanted constant arrays, so that $x[e]$ could denote a value instead of a location, we would define arrays to have type $\texttt{Bv} \to \texttt{Dv}$.

Exercise 40 Give examples that demonstrate the advantages of arrays of type $\texttt{Bv} \to \texttt{Dv}$. □

6.3.1 Pointers

In a language with pointers, locations are denotable, expressible, and storable:

$$\begin{aligned}
\texttt{Dv} &= \texttt{Loc} + \cdots \\
\texttt{Ev} &= \texttt{Loc} + \cdots \\
\texttt{Sv} &= \texttt{Loc} + \cdots
\end{aligned}$$

We need syntax for dereferencing and, in languages like C, an "address of" operation that provides a reference as an rvalue. Here we use prefix `*` for dereferencing and prefix `&` for referencing, as in C.

$$\begin{aligned}
e &= \texttt{*}x \mid \texttt{\&}x \mid \cdots \\
\mathcal{L}[\![\texttt{*}\ x]\!] &= \mathcal{R}[\![x]\!] \\
&= \lambda\rho\sigma\ .\ \sigma(\rho[\![x]\!]) \\
\mathcal{R}[\![\texttt{*}\ x]\!] &= \lambda\rho\sigma\ .\ \sigma(\mathcal{R}[\![x]\!]) \\
&= \lambda\rho\sigma\ .\ \sigma(\sigma(\rho[\![x]\!])) \\
\mathcal{L}[\![\texttt{\&}\ x]\!] &= \lambda\rho\sigma\ .\ \mathit{error} \\
\mathcal{R}[\![\texttt{\&}\ x]\!] &= \lambda\rho\sigma\ .\ \rho[\![x]\!]
\end{aligned}$$

Exercise 41 In C, arrays and pointers are treated in the same way: the expressions $a[i]$ and $\texttt{*}(a + i)$ are considered to be synonyms. Explain this equivalence by means of semantic equations. □

6.4 Side Effects

We have distinguished *expressions* that yield a value and *commands* that change the state. Many languages provide constructs that do both at the same time. We can refer to such constructs as either *commands that return values* or *functions that have side effects.* In Pascal, the body of a function is a command that may change non-local state; thus a function call in Pascal is an expression with (possible) side effects. In C, constructs such as assignments are called *expression statements* because they may be used either as expressions with side effects or as statements (commands in our terminology). If fact, if e is any C expression, then "e;" is a C statement that yields the value of e.

There are two ways in which we could admit side effects into the semantics. The first way is to add a new kind of semantic function to the functions that we have already:

$$\begin{array}{lrcl}
\text{Expressions:} & \mathcal{V}[\![\cdot]\!] & : & \texttt{Exp} \to \texttt{Env} \to \Sigma \to \texttt{Ev} \\
\text{Commands:} & \mathcal{C}[\![\cdot]\!] & : & \texttt{Com} \to \texttt{Env} \to \Sigma \to \Sigma \\
\text{Side-effects:} & \mathcal{M}[\![\cdot]\!] & : & \texttt{Syn} \to \texttt{Env} \to \Sigma \to (\Sigma \times \texttt{Ev})
\end{array}$$

Note the new syntactic domain `Syn` which is intended to include functions with side effects and/or commmands that return values. The new function $\mathcal{M}$ takes a command, an environment, and a state, and returns a pair consisting of a new state and a value.

This approach would be suitable for Pascal, which makes limited use of side-effects. Even in Pascal, however, the side-effects are rather insidious. Although simple expressions, such as numbers, would remain in `Exp`, expressions such as $e_1 + e_2$ would move to `Syn` because one or both of the arguments of $+$ might be function calls.

An alternative approach is to abandon the semantic functions $\mathcal{V}$ and $\mathcal{C}$ and use $\mathcal{M}$ for everything. This approach works for any language, but it works particularly well for languages such as C in which side effects are used extensively.

The design of C was strongly influenced by Algol68; in fact, C features such as cast, coercion, references, conditional expressions, function declarations, the type `void`, and expression statements are all based on Algol68. Algol68 is a highly uniform language, in which *all* constructs have a value. If the value is not useful, it is assigned the type `void`, a type with a single value that requires $\log_2 1 = 0$ bits of storage.

In this section, we will use $\mathcal{M}$ only. Since all syntactic constructs are required to return a state and a value, we introduce a dummy value ω for commands that don't have anything useful to return. For example:

$$\mathcal{M}[\![\texttt{skip}]\!] \quad = \quad \lambda\rho\sigma\,.\,(\sigma, \omega).$$

6.5 Blocks with Values

The simplest construct that both changes the state and yields a value is a *block-expression*. (A block-expression corresponds to the body of a function in Pascal.) The syntax of a block expression is

$$\{\, d; c; e \,\}.$$

Informally, the meaning of a block expression is: evaluate the declarations d, evaluate the commands c, and yield the value of the expression e.

The concrete syntax of block expression varies between languages. In Algol68, the expression alone is sufficient. In Pascal, block expressions are allowed only as function bodies and the value is returned by a statement of the form $f := e$, where f is the name of the function. In C, the `return` statement is used to yield the value of the block (and also, of course, to transfer control out of the block).

The meaning of a value block is given by

$$\begin{array}{rcll}
\mathcal{M}[\![\{\, d; c; e \,\}]\!] & = & \lambda\rho\sigma\,.\,\texttt{let} & \rho' = \mathcal{D}[\![d]\!]\,\rho \\
 & & \texttt{in} & \texttt{let}\ \ (\sigma', v) = \mathcal{M}[\![c]\!]\,\rho'\,\sigma \\
 & & & \texttt{in}\ \ \ \mathcal{M}[\![e]\!]\,\rho'\,\sigma'
\end{array}$$

In words:

- ▷ Evaluate the declarations d in environment ρ, giving a new environment ρ'.
- ▷ Evaluate the commands c in environment ρ' and state σ, giving a new state σ' and yielding a value v (which is discarded).
- ▷ Evaluate expression e in environment ρ' and state σ', giving a new state and yielding a value.

6.6 Procedures and Functions

In principle, a procedure declaration gives a name to a command and a function declaration gives a name to an expression. Assume for now that procedures and functions do not have parameters. We could add procedures and functions to L_2 as follows.

- ▷ Add procedure and function declarations to `Dec`, using p as a typical procedure name and f as a typical function name.

$$d \quad = \quad \texttt{proc}\, p = c \mid \texttt{fun}\, f = e \mid \cdots$$

- ▷ Add procedure calls to `Com` and function calls to `Exp`:

$$\begin{aligned} c &= \texttt{call}\, x \mid \cdots \\ e &= \texttt{call}\, x \mid \cdots \end{aligned}$$

- ▷ Extend the environment so that it can store procedure and function declarations:

$$\texttt{Dv} \quad = \quad \texttt{Loc} + \Sigma \to \texttt{Ev} + \Sigma \to \Sigma.$$

- ▷ Use the same semantic functions as before:

$$\begin{array}{lll} \text{Declarations:} & \mathcal{D}[\![\cdot\ ]\!] & : \texttt{Dec} \to \texttt{Env} \to \texttt{Env} \\ \text{Expressions:} & \mathcal{V}[\![\cdot\ ]\!] & : \texttt{Exp} \to \texttt{Env} \to \Sigma \to \texttt{Ev} \\ \text{Commands:} & \mathcal{C}[\![\cdot\ ]\!] & : \texttt{Com} \to \texttt{Env} \to \Sigma \to \Sigma \end{array}$$

- ▷ Provide meanings for declarations and calls.

$$\begin{aligned} \mathcal{D}[\![\texttt{proc}\, p = c]\!] &= \lambda\rho\,.\,\rho \oplus \langle p = \mathcal{C}[\![c\ ]\!]\ \rho\rangle \\ \mathcal{D}[\![\texttt{fun}\, f = e]\!] &= \lambda\rho\,.\,\rho \oplus \langle f = \mathcal{V}[\![e]\!]\ \rho\rangle \\ \mathcal{C}[\![\texttt{call}\, p]\!] &= \lambda\rho\sigma\,.\,\rho[\![p]\!]\,\sigma \\ \mathcal{V}[\![\texttt{call}\, f]\!] &= \lambda\rho\sigma\,.\,\rho[\![f]\!]\,\sigma \end{aligned}$$

The semantics of procedures and functions are so similar that it makes more sense to treat the bodies of both as block-expressions, as defined in the previous section. Rather than writing `fun` f, we will write $f()$ to denote a function abstraction, as in C. We have:

$$\begin{aligned} \mathcal{D}[\![f() = e]\!] &= \lambda\rho\,.\,\rho \oplus \langle f = \mathcal{M}[\![e]\!]\ \rho\rangle \\ \mathcal{M}[\![f\ ()]\!] &= \lambda\rho\sigma\,.\,\rho[\![f\ ]\!]\,\sigma \end{aligned}$$

When $\rho[\![f]\!]\sigma$ is evaluated, $\rho[\![f\ ]\!]$ is replaced by $\mathcal{M}[\![e\ ]\!]\rho$. Note, however, that this ρ is the environment in which the function was declared, not the environment in which it is executed.

Note also the types involved:

$$\begin{array}{rcl} c & : & \mathtt{Com} \\ \mathcal{M}[\![c]\!] & : & \mathtt{Env} \to \Sigma \to (\Sigma \times \mathtt{Ev}) \\ \mathcal{M}[\![c]\!]\,\rho & : & \Sigma \to (\Sigma \times \mathtt{Ev}) \\ \mathcal{M}[\![c]\!]\,\rho\,\sigma & : & \Sigma \times \mathtt{Ev} \end{array}$$

The value stored in the environment by the declaration is $\mathcal{M}[\![c]\!]\,\rho$, which we can consider as a "state transformer", $\Sigma \to (\Sigma \times \mathtt{Ev})$. When we call the procedure, we apply this "state transformer" to the current state, giving a new state and a value. The procedure declaration is evaluated in the environment of the declaration, and the call is evaluated in the current run-time state. This corresponds to the **static scoping** rules of languages such as Pascal and C.

6.7 Parameters

Things get interesting when we introduce procedures and functions with parameters. First, a convention.

Definition 42 *A* **parameter** *is a variable name used in the declaration of a procedure or function to stand for the argument that will eventually be supplied to the function.*

Definition 43 *An* **argument** *is an expression that is passed to a procedure or function when the procedure or function is called.*

Parameters are also called "formal parameters" and "dummy arguments". Arguments are also called "actual parameters" and "actual arguments". The one-word conventions seem simpler.

For simplicity, let us consider functions of one argument. The syntax is

$$\begin{array}{rcl} d & = & f(x) = e \mid \cdots \\ c & = & f(e) \mid \cdots \end{array}$$

The semantic equations are

$$\begin{array}{rcl} \mathcal{D}[\![f\,(x) = e]\!] & = & \lambda\rho \,.\, \rho \oplus \langle f = \lambda a \,.\, \mathcal{M}[\![e]\!]\,\rho \oplus \langle x = a\rangle\rangle \\ \mathcal{M}[\![f(e)]\!] & = & \lambda\rho\sigma \,.\, \rho[\![f]\!]\,(\mathcal{V}[\![e]\!]\rho\,\sigma)\,\sigma \end{array}$$

The declaration stores an *abstract* in the environment. The type of $\mathcal{M}[\![e]\!]\,\rho$ is $\Sigma \to (\Sigma \times \mathtt{Ev})$ and we assume that the type of the argument will be $\mathtt{Ev}$. The type of the abstract, $f = \lambda x \,.\, \mathcal{M}[\![e]\!]\,\rho$, is therefore $\mathtt{Ev} \to \Sigma \to (\Sigma \times \mathtt{Ev})$. Evaluation of the call $f(e)$ proceeds as follows.

- ▷ Look up the function f in the environment, obtaining an abstract $\lambda a \,.\, \mathcal{M}[\![e]\!]\,\rho \oplus \langle x = a\rangle$.
- ▷ Evaluate the argument e in the current state, obtaining $b = \mathcal{V}[\![e]\!]\,\rho\,\sigma$.
- ▷ Apply the abstract to the argument:

$$\begin{array}{rl} & (\lambda a \,.\, \mathcal{M}[\![e]\!]\,\rho \oplus \langle x = a\rangle)(b) \\ = & \mathcal{M}[\![e]\!]\,\rho \oplus \langle x = b\rangle. \end{array}$$

▷ Evaluate the result in the current state, σ.

For example, consider the program

$$\begin{array}{l} f(x) = x + 1; \\ f(2) \end{array}$$

Assuming an empty initial environment, evaluating the declaration gives

$$\begin{array}{rcl} \rho & = & \mathcal{D}\,[\![f\ (x) = x + 1]\!] \\ & = & \langle f = \lambda a\,.\,\mathcal{M}\,[\![x\ + 1]\!]\,\langle x = a \rangle\rangle \end{array}$$

Then, for an arbitrary state σ:

$$\begin{array}{rclll} & & \text{Function} & \text{Argument} & \text{State} \\ \mathcal{M}\,[\![\,f(2)\,]\!]\ \rho\,\sigma & = & \rho(f) & (\mathcal{V}\,[\![\,2\,]\!]\ \rho\,\sigma) & \sigma \\ & = & (\lambda a\,.\,\mathcal{M}\,[\![x\ + 1]\!]\,\langle x = a\rangle) & (2) & \sigma \\ & = & \mathcal{M}\,[\![\,x + 1\,]\!]\,\langle x = 2\rangle & & \sigma \\ & = & (\sigma, 3) & & \end{array}$$

The semantics that we have given is that of **call by value**: the function receives the value of the argument, as in C. To model **call by reference** (corresponding to `var` parameters in Pascal), evaluation of the argument would yield a location, and the abstract would be parameterized with respect to this location.

Exercise 42 Write semantic equations for functions and procedures with call by reference parameters. □

6.8 Continuations

Older languages typically have commands for direct transfer of control, usually called `goto` commands. Denotational semantics models such transfers of control by means of **continuations**. Continuations are actually a very powerful and general mechanism that can do much more than provide a denotation for `goto` commands. In this section, however, we will use them to describe a simple language with `goto`.

A continuation is a state transformer. In a continuation semantics, each command passes a continuation to its successor. Intuitively, the continuation is the "rest of the program". For simplicity, we will use just the following semantic domains:

$$\begin{array}{rcccl} \kappa & \in & \texttt{Cont} & = & \Sigma \to \Sigma \\ \rho & \in & \texttt{Env} & = & \texttt{Lab} \to \texttt{Cont} \end{array}$$

A typical continuation is named κ and its type is $\Sigma \to \Sigma$. The environment maps labels in `Lab` to continuations. (The environment must also map variables to their locations, as usual, but we ignore this for now.)

The semantic function for commands requires an environment, a continuation, and a state, and it yields a new state:

$$\mathcal{C}\,[\![\,\cdot\,]\!] \quad : \quad \texttt{Com} \to \texttt{Env} \to \texttt{Cont} \to \Sigma \to \Sigma.$$

$$
\begin{array}{rcl}
\mathcal{C}[\![\,\texttt{skip}\,]\!]\rho\,\kappa\,\sigma & = & \kappa\,\sigma \\
\mathcal{C}[\![\,x := e\,]\!]\rho\,\kappa\,\sigma & = & \kappa(\sigma \oplus \langle \rho[\![x]\!] = \mathcal{V}[\![e]\!]\sigma\rangle) \\
\mathcal{C}[\![\,c_1; c_2\,]\!]\rho\,\kappa\,\sigma & = & \mathcal{C}[\![\,c_1\,]\!]\,\rho(\mathcal{C}[\![\,c_2\,]\!]\rho\,\kappa\,)\,\sigma \\
\mathcal{C}[\![\,\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\,]\!]\rho\,\kappa\,\sigma & = & \mathcal{V}[\![\,e\,]\!]\sigma \rightarrow \mathcal{C}[\![c_1]\!]\rho\,\kappa\,\sigma,\ \ \mathcal{C}[\![\,c_2\,]\!]\rho\,\kappa\,\sigma \\
\mathcal{C}[\![\texttt{goto}\ \ \ell\,]\!]\rho\,\kappa\,\sigma & = & \rho[\![\ell]\!]\sigma \\
\mathcal{C}[\![\,\ell : c\,]\!] & = & \mathcal{C}[\![\,c\,]\!] \\
\mathcal{C}[\![\ell_1 : c_1;\ \ell_2 : c_2;\ \ldots;\ \ell_n : c_n\,]\!]\rho\,\kappa & = & \texttt{let}\ \ \begin{array}[t]{rcl} \kappa_1 & = & \mathcal{C}[\![\,c_1\,]\!]\rho\,'\,\kappa_2 \\ \kappa_2 & = & \mathcal{C}[\![\,c_2\,]\!]\rho\,'\,\kappa_3 \\ & & \ldots\ldots \\ \kappa_n & = & \mathcal{C}[\![\,c_n\,]\!]\rho\,'\,\kappa \\ \rho' & = & \rho \oplus \langle \ell_1 = \kappa_1, \ldots, \ell_n = \kappa_n\rangle \end{array} \\
& & \texttt{in}\ \ \ \kappa_1
\end{array}
$$

Figure 26: Continuation Semantics

The semantic equations are shown in Figure 26.

The `skip` command, instead of returning the state unchanged, applies its continuation to the state. Similarly, the assignment command applies its continuation to its state updated with the new value of the assigned variable. Structured commands use their additional argument in the obvious way.

The statement `goto` ℓ transfers control to a *labelled command* of the form $\ell : c$. The meaning of $\ell : c$ is the same as the meaning of c.

The key part of the semantics is in the last equation, which gives the meaning of a compound command in which each component has a label. There is a series of continuations, each of which includes its predecessor, and a new environment, ρ', which associates each label to the corresponding continuation.

To see how this works, consider the following program, which we refer to as p.

$\ell_1 : x := 0;$
$\ell_2 : x := x + 1;$
$\ell_3 :$ `if` $x < 2$ `then goto` ℓ_2 `else skip`

We evaluate this program with an empty environment ($\rho = \langle\rangle$), the identity function as continuation ($\kappa = I$), and an empty state ($\sigma = \langle\rangle$). That is, we must evaluate

$$\sigma' \ \equiv\ \mathcal{C}[\![\,p\,]\!]\langle\rangle\ I\langle\rangle.$$

Using the rule for labelled compound statements above, we have:

$$
\begin{array}{rcl}
\kappa_1 & = & \mathcal{C}[\![\,x := 0\,]\!]\,\rho\,\kappa_2 \\
\kappa_2 & = & \mathcal{C}[\![\,x := x + 1\,]\!]\,\rho\,\kappa_3 \\
\kappa_3 & = & \mathcal{C}[\![\,\texttt{if}\ x < 2\ \texttt{then}\ \ \texttt{goto}\ \ell_2\ \texttt{else skip}\,]\!]\,\rho I \\
\rho & = & \langle \ell_1 = \kappa_1, \ell_2 = \kappa_2, \ell_3 = \kappa_3\rangle
\end{array}
$$

We use these values several times in the following simplification.

$$
\begin{aligned}
\sigma' &= \mathcal{C}\,[\![\, p \,]\!]\langle\rangle\; I\langle\rangle \\
&= \kappa_1\langle\rangle \\
&= \mathcal{C}\,[\![\, x := 0 \,]\!]\,\rho\,\kappa_2\langle\rangle \\
&= \kappa_2\langle x = 0\rangle \\
&= \mathcal{C}\,[\![\, x := x + 1 \,]\!]\,\rho\,\kappa_3\langle x = 0\rangle \\
&= \kappa_3\langle x = 1\rangle \\
&= \mathcal{C}\,[\![\, \texttt{if}\ x < 2\ \texttt{then goto}\ \ell_2\ \texttt{else skip} \,]\!]\rho I\ \langle x = 1\rangle \\
&= \mathcal{C}\,[\![\, \texttt{goto}\ \ell_2 \,]\!]\,\rho I\langle x = 1\rangle \\
&= \rho\,[\![\, \ell_2 \,]\!]\,\langle x = 1\rangle \\
&= \kappa_2\langle x = 1\rangle \\
&= \mathcal{C}\,[\![\, x := x + 1 \,]\!]\,\rho\,\kappa_3\langle x = 1\rangle \\
&= \kappa_3\langle x = 2\rangle \\
&= \mathcal{C}\,[\![\, \texttt{if}\ x < 2\ \texttt{then goto}\ \ell_2\ \texttt{else skip} \,]\!]\rho I\ \langle x = 2\rangle \\
&= \mathcal{C}\,[\![\, \texttt{skip} \,]\!]\,\rho I\langle x = 2\rangle \\
&= I\langle x = 2\rangle \\
&= \langle x = 2\rangle
\end{aligned}
$$

As mentioned above, continuations can be used to describe control structures that are considerably more sophisticated than `goto` commands. Consider the following `C` program:

```
#include <setjmp.h>
jmp_buf state;
{
   .....
   if (!setjmp(state)) {
     /* normal processing */
     .....
     longjmp(state, 3);
     .....
     }
   else {
     /* exception processing */
   }
}
```

The first time through, `setjmp(state)` returns 0 and "normal procesing" begins. If the `longjmp` command is never executed, the program complets "normal processing" and stops. If the `longjmp` command is executed, the effect is as if `setjmp(state)` "returned" for a second time, yielding 3, thereby executing "exception processing". (The value 3 could be passed to the exception processing block, of course, although we have not done so in this example.) Note that `longjmp` does not necessarily appear in the scope of the `setjmp` block as shown here: it could be deeply nested in functions called from this block.

The `setjmp/longjmp` mechanism is simple to implement: `setjmp` stores the current values of the program counter and stack pointer (and perhaps some other stuff, such as register contents) in the variable `state`, and `longjmp` retrieves those values to restore the state in which `setjmp` was executed.

We could give a semantics for `setjmp/longjmp` using continuations as follows: `setjmp` is a command that stores its continuation in memory and assigns the memory location to a variable; `longjmp` is a command that retrieves the continuation from memory and applies it to the current state.

Exercise 43 Many languages have a command that exits a function or procedure: for example, `return` in C. Show how the semantics of such a command can be expressed using continuations. Show further that a command such as C's `return` e can also be expressed using continuations. □

6.9 Summary

In Sections 4, 5, and the present section, we have seen how to use denotational semantics to describe language features, including some quite arcane features. It would be a straightforward, though very tedious exercise, to write a denotational semantics for a language such as Pascal. We could even write a denotational semantics for most of C, if we omitted the preprocessor.

Of course, the language that we could describe in this way would be an idealized form of the actual language that is implemented. For example, the semantics uses domains such as natural numbers with unbounded values, but a processor provides only a small subset of the integers. (We can implement "bignums", of course, but we are still limited by bounded memory.) We have not discussed real numbers in the semantics, but it is clear that there is a vast gulf between the mathematical set $\mathbb{R}$ of real numbers and the peculiar subset of the rationals provided by floating-point processors. Even worse, computer arithmetic does not satisfy the most fundamental mathematical laws, such as associativity. Nevertheless, it is useful to be able to give a precise meaning to each program, even if that meaning is somewhat idealized.

The denotational semantics of a simple practical language such as Pascal is daunting. Writing the semantics of a modern production language is a huge undertaking: the semantics of Ada occupies a sizable book. Why are semantics so complex? Ashcroft and Wadge (1982) have provided an explanation: actual languages are designed *operationally* — the question "How will they implement this?" is never far from a designer's mind. One of the lessons of denotational semantics is that concepts that are operationally simple may be semantically complex and *vice versa*.

As a simple example, consider the `while` and `repeat` statements. Operationally, the `repeat` statement is simpler than the `while` statement because it requires only one conditional jump (Assignment 1, question 3). For this reason, it is preferred by assembly language progammers. At the language level, however, the `while` statement is simpler to understand because the condition is evaluated first. Furthermore, the `while` statement has a useful property that the `repeat` statement lacks: the body of the loop may never be executed at all.

A more extreme example is the `goto` statement. Jumps are easily explained and implemented operationally, but an abstract description of their behaviour requires the complex machinery of continuations.

Considerations such as these suggest an alternative way of using denotational semantics: we could use semantic principles to guide the design of programming languages. The resulting languages might be hard to implement, but they should be simple to understand and to use.

The first example of this approach was LISP, designed by John McCarthy (McCarthy 1960; McCarthy, Abrahams, Edwards, Hart, and Levin 1962; McCarthy 1963; McCarthy 1981). The basic idea of LISP was that the programmer would write a set of mutually recursive function definitions and the processor would attempt to find a value of a particular expression $f(a_1, \ldots, a_n)$ consistent with these equations (note that this is actually a fixed-point calculation). This is a very simple *semantic* concept and it is interesting to note that the original LISP interpreter (the operational implementation of the semantics) had a fundamental error from which LISP has never recovered: the incorrect implementation became the *de facto* definition of the semantics.

Aside The following Pascal program would print "`Pascal`" (static scoping) but the corresponding LISP program would print "`LISP`" (dynamic scoping). Dynamic scoping has its uses, but it was not McCarthy's intended semantics.

```
var S : String
procedure P;
   begin write (S) end;
procedure Q (S : String);
   begin P end;
begin
   S := 'Pascal';
   Q ('LISP')
end
```

In the next section, we will look at ways in which denotational semantics can help us to design languages.

7 Applying Semantics to Language Design

We begin this section by reviewing some comparisons of the operational and denotational views of languages. We then consider ways in which concepts from semantics might be used to improve existing languages or to define better languages.

7.1 Operational *versus* Denotational Semantics

The thesis of Ashcroft and Wadge (1982) is that programming languages *are* designed operationally but *should be* designed denotationally. In this section, we consider several examples of this distinction.

7.1.1 Loop Statements

We have already seen that there is a small difference between `while` and `repeat` loops. For the operational semantics, we have:

$$\begin{aligned}
\mathcal{C}\,[\![\texttt{while}\ b\ \texttt{do}\ c]\!] &\rightarrow \langle\texttt{lab}\ \alpha\rangle;\mathcal{C}\,[\![b]\!];\langle\texttt{jz}\ \beta\rangle;\mathcal{C}\,[\![c]\!];\langle\texttt{j}\ \alpha\rangle;\langle\texttt{lab}\ \beta\rangle\\
\mathcal{C}\,[\![\texttt{repeat}\ c\ \texttt{until}\ b]\!] &\rightarrow \langle\texttt{lab}\ \alpha\rangle;\ \mathcal{C}\,[\![c]\!];\ \mathcal{C}\,[\![b]\!];\ \langle\texttt{jz}\ \alpha\rangle;
\end{aligned}$$

Viewed in this light, `repeat` is simpler than `while` because it contains one jump rather than two and, for the same reason, it is more efficient. In fact, a compilers commonly transform

$$\texttt{while}\ b\ \texttt{do}\ c$$

to

$$\texttt{if}\ b\ \texttt{then}\ (\texttt{repeat}\ c\ \texttt{until}\ \neg b)\ \texttt{else skip}$$

Using denotational semantics, however, we have:

$$\begin{aligned}
\mathcal{C}\,[\![\texttt{while}\ e\ \texttt{do}\ c]\!] &= \mu X\,.\,\lambda\sigma\,.\,\mathcal{V}\,[\![e]\!]\,\sigma \rightarrow (X \circ \mathcal{C}\,[\![c]\!])\sigma,\ \sigma\\
\mathcal{C}\,[\![\texttt{repeat}\ c\ \texttt{until}\ b]\!] &= \mu X\,.\,\lambda\sigma\,.\,\texttt{let}\ \sigma' = \mathcal{C}\,[\![c]\!]\ \sigma\ \texttt{in}\ \mathcal{V}\,[\![b]\!]\sigma' \rightarrow \sigma', X\sigma'
\end{aligned}$$

These equations show that, whereas the meaning of the `while` statement can be understood in terms of the initial state, σ, the meaning of the `repeat` statement can be understood only in terms of the modified state, σ', obtained by executing the command c. This additional complexity also appears in the axiomatic semantics.

Intuitively, it is easier to reason about `while` statements than about `repeat` statements. The denotational semantics corresponds to our intuition, but the operational semantics does not.

Exercise 44 Give operational and denotational semantics for a loop-with-exit control structure such as C's

```
    while (1) { .....  break; .....  }
```

□

7.1.2 The `goto` Command

We can apply a similar kind of reasoning to `goto` commands. The operational semantics of `goto` for the abstract machine of Section 2.3 is straightforward:

$$\begin{aligned} \mathcal{C}\,[\![\, \ell :]\!] &\;\to\; \langle \mathtt{lab}\ \ell \rangle \\ \mathcal{C}\,[\![\, \mathtt{goto}\ \ell \,]\!] &\;\to\; \langle \mathtt{j}\ \ell \rangle \end{aligned}$$

But, as we have seen in Section 6.8, the denotational semantics of `goto` commands requires the introduction of continuations and a complete rewriting of the semantics.

Once again, our intuition is in line with the denotational semantics: programs with `goto` statements are hard to understand[3] compared to programs that use structured commands such as `while` and `repeat`.

When we attempt to describe commands such as `goto` denotationally, the semantic equations lose both their elegance and their value. It is hard to reason with continuation semantics and the equations increasingly resemble an inefficient interpreter.

7.1.3 Functions and Procedures

Functions and procedures, as defined in most languages, are straightforward to implement (that is, they have a simple operational semantics). Arguments and a link are placed on the stack and control is transferred to the code of the function; when the function has completed its computations, it places a result on the stack and uses the link to return control to the caller.

In practice, however, this simple mechanism leads to a hodge-podge of gadgets and to the omission of simple but useful language features (we will discuss this further in Section 7.3).

There is a simple and clear semantic distinction between a function and a procedure: a function is a named expression that returns a value and a procedure is a named command that has an effect. The call-return mechanism confuses this distinction, making it appear that functions and procedures are both varieties of command, but functions happen to return a value. Moreover, since the call-return mechanism does not work for certain kinds of function (such as functions returning functions), languages typically do not provide these kinds of function.

7.1.4 Passing Arguments

Most programming language pass arguments *by value*: the argument is evaluated, and then its value is passed to the function. An alternative approach, required by Algol60 but rarely used since, is to pass arguments *by name*: intuitively, the text of the argument, rather than its value, is passed to the function.

In λ-calculus, there is sometimes a choice of several β-reductions that we can use to simplify an expression. Consistently choosing the *innermost* redex corresponds to call by value; consistently choosing the *outermost* redex corresponds to call by name.

Consider the rather contrived function

[3] Although perhaps not quite as hard as semantic equations with continuations!

```
function f (x, y: integer): integer;
   begin
     if x = 0 then f := −1 else f := y
   end
```

and the invocation $e = f(z, 1/z)$. If we use call by value when $z = 0$, the function is never called because evaluation of the arguments fails. If we use call by name, the expression effectively becomes

```
if z = 0 then e := −1 else e := 1/z
```

which is well-defined when $z = 0$. (It was considerations such as these that led the Algol60 committee to require call by name as the default method of passing arguments.)

Call by name is usually implemented by means of thunks: a *thunk* is a function that evaluates the argument. The caller places thunks on the stack and the function calls them when necessary. In the example above, there would be a thunk for $1/z$ on the stack and it would be called only if $z \neq 0$. If the thunk does not have side-effects, the compiler can optimize the program by replacing the thunk by its value after it has been called for the first time.

Once again, we observe that call by value is easy to understand operationally and to implement, but call by name is simpler semantically (because it corresponds to conventional substitution) but harder to implement efficiently.

Exercise 45 What is the value of the Algol60 expression $f(1/n, n, 1, 5)$? Assume that f is the Algol60 function defined below and that arguments are passed by name. (The technique used here is called *Jensen's device.*)

```
real procedure f (name p, var q, val r, s);
   begin real p, t; int q, r, s;
     t := 0; q := r;
     while q ≤ s do
       begin t := t + p; q := q + 1
       end;
     f := t
   end;
```

□

7.1.5 Data Structures

Although we have not explored the semantics of data structure in detail, we can nevertheless note similar kinds of tension between operational and denotational approaches.

Consider arrays first. The array is essentially an operational construct, being a straightforward abstraction of a memory with a linear sequence of addresses. Arrays are easy and efficient to implement: $a[i]$ denotes the address $\mathcal{V}[\![a]\!]\,\rho + k \times \sigma(\mathcal{V}[\![i]\!]\,\rho)$, in which $\mathcal{V}[\![a]\!]\,\rho$ is the base address of the array, $\sigma(\mathcal{V}[\![i]\!]\,\rho)$ is the value of the index, and k is the component size.

The problem with arrays is that the assignment $a[i] := e$, which we think of as updating a component, should be treated as yielding a new array. To see the problems that arise with the "update component" model, we return to weakest-precondition semantics, in which

$$\mathcal{W}(x := e, Q) \quad = \quad Q[x \leftarrow e].$$

This says that the weakest precondition that will ensure Q after executing $x := e$ is $Q[x \leftarrow e]$. (Read $x \leftarrow e$ as "with x replaced by e: use of the left arrow avoids confusion with equality below.) Applying this to an array component, we have

$$\begin{aligned} \mathcal{W}(a[i] := 5, a[i] = 5]) &\iff (a[i] = 5)[a[i] \leftarrow 5] \\ &\iff 5 = 5 \\ &\iff \mathit{true}. \end{aligned}$$

That is, for any precondition (apart from *false*, of course), the assignment $a[i] := 5$ ensures that $a[i] = 5$, as we would expect. By similar reasoning,

$$\begin{aligned} \mathcal{W}(a[i] := 5, a[i] = a[j]) &\iff (a[i] = a[j])[a[i] \leftarrow a[j]] \\ &\iff a[j] = a[j] \\ &\iff \mathit{true}. \end{aligned}$$

But this is wrong because, if $i = j$, the assignment does not change the value of $a[i]$. We must add an additional step in the reasoning for the case $i = j$.

$$\begin{aligned} \mathcal{W}(a[i] := 5, a[i] = a[j]) &\iff (a[i] = a[j])[a[i] \leftarrow a[j]] \\ &\iff (i = j \wedge 5 = 5) \vee (i \neq j \wedge a[j] = 5) \\ &\iff i = j \vee (i \neq j \wedge a[j] = 5). \end{aligned}$$

This example is trivial, but it demonstrates the care we must use in reasoning about arrays. It is an example of the more general problem of *aliasing*: two expressions denoting the same address.

In programs without functions or procedures, we can safely assume that variables with distinct names have distinct addresses. In fact, we must assume this if we are to have a simple semantic rule even for the simple assignment $x := y$. In a block-structured language, however, functions and procedures introduce the possibility that a local variable could have the same address as a non-local variable: this possibility greatly complicates a full treatment of nested scopes. The same problem occurs with arrays, because there is always a possibility that $a[i]$ and $a[j]$ may denote the same address.

Functional programming languages (discussed further in the next section) do not permit the alteration of values by assignment. Arrays are updated by a function call *update* (a, i, x) which means, roughly, "like array a but with $a[i] = x$". Figure 27 shows a simple implementation of functional arrays. The array a contains the first nine digits of π. The array b is the same but has $a[4] = 7$; that is, $b = \mathit{update}\,(a, 4, 7)$. Similarly, $c = \mathit{update}\,(b, 2, 9)$. Conceptually, b and c are new arrays, but the implementation simply creates "array update cells" tagged to distinguish them from true arrays (the small circles in Figure 27 indicate the tags).

A practical implementation uses a tree structure rather than a list for the update cells, giving $\mathcal{O}(\log N)$ access time in the worst case. If the number of updates become large, or the original array becomes garbage, the run-time system constructs the updated arrays.

Whereas arrays are easy to implement and hard to reason about, lists are hard to implement but easy to reason about. This is why the first functional and logical programming languages provided lists but not arrays. Figure 28 shows a typical list structure. The original list is a. The function call *car* (a) returns x, the first component of the list, and the call *cdr* (a) returns b, the tail of the list. (The function names are borrowed from LISP.) We can extend the list

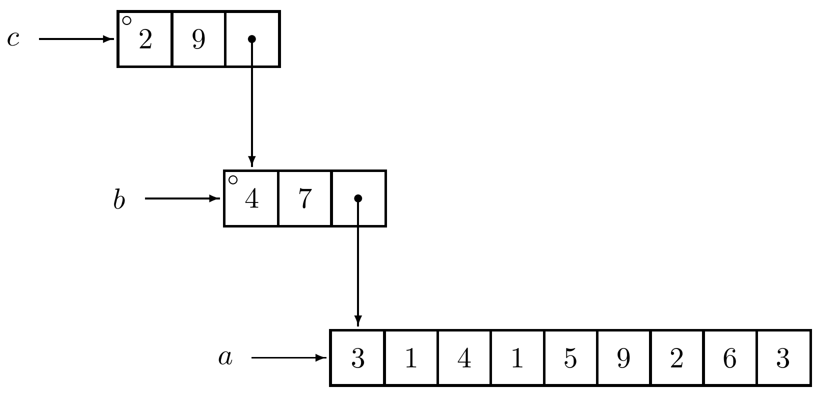

Figure 27: Functional Arrays

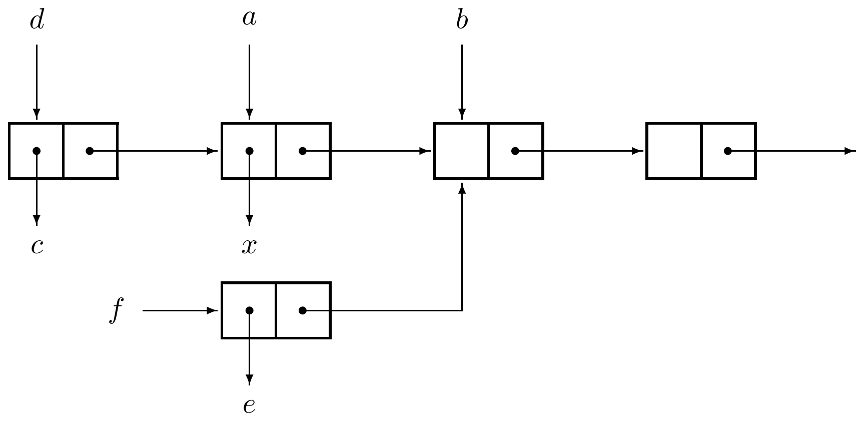

Figure 28: List Processing

a with a new cell using $d = \mathit{cons}\,(c, a)$. We can even obtain a list with a different head, e, but the same tail as a using $f = \mathit{cons}\,(e, \mathit{cdr}\,(a))$.

The important feature of all of these operations is that they are pure functions: none of them changes the original list, a. Thus reasoning about list operations is straightforward: all the usual algebraic manipulations are appropriate. The downside is that each *cons* call allocates memory and a garbage collector is required to reclaim memory.

7.1.6 Functional Programming

The examples above contrast operational simplicity with semantic complexity. The following examples show the other side of the coin: semantic simplicity leading to operational complexity.

The idea of functional programming is very simple. As conceived by McCarthy (1960), a functional program evaluates a function call $f(\overline{x})$, in which $\overline{x}$ is a vector of arguments, in the context of a set of mutually recursive definitions

$$f_i \quad = \quad \lambda\overline{x} \,.\, e_i \qquad (1 \le i \le n)$$

The meaning of such a program is a fixed point in a suitable domain. Practical functional languages permit functions to be arguments and results of other functions.

Although they have simple semantics, functional languages have developed slowly because they are hard to implement efficiently. Since values of variables never change, each function application requires a fresh set of variables. Since processors have finite memory, functional languages require garbage collectors.[4] Early implementations of functional languages such as LISP were notorious for the delays caused by garbage collection.

The storage problem is partly solved by using a stack: arguments are pushed on to the stack when a function is called and popped from the stack when the function returns. This works for functions that pass functions as arguments, but not for functions that return functions as results. Functional composition, for example, cannot be defined in a stack-based language.

7.1.7 Logic Programming

Logic programming provides another combination of simple semantics and hard implementation. Ideally, a logic program has the form

$$\forall \overline{x} \,.\, \exists \overline{y} \,.\, P(\overline{x}, \overline{y})$$

and executing the program consists of reading values for $\overline{x}$ and finding values for $\overline{y}$ for which P is true. This approach does not work because the predicate calculus is undecidable. Languages such as Prolog permit only Horn clauses in P because there is a semi-decidable proof system for Horn clauses (if there are solutions, the Prolog system will find them in finite time but, if there are no solutions, the program may not terminate).

A correct Prolog interpreter explores a large proof tree looking for solutions. The "cut" operator, however, allows programmers to prevent exploration of parts of the tree known not to contain solutions. The cut operator is simple to implement (it is similar to non-local exit in a conventional programming language) but hard to describe semantically. Worse yet, incorrect usage may prevent the program from finding valid solutions.

Prolog interpreters use unification to match formulas. For example, Prolog uses the substitution $\langle x = H(A, z), y = A \rangle$ to unify the formulas $F(A, G(x, B))$ and $F(y, G(H(A, x), B))$. Actually, these formulas should not unify, because they contain x in positions that do not correspond, but most Prolog interpreters do not perform the "occurs check" because it is inefficient. Consequently, it is theoretically possible for a Prolog interpreter to "prove" statements that are not true — in the name of efficiency!

7.2 The Principle of Correspondence

Consider the following two code fragments:

```
var x = e;            proc P(x)
write x                   { write x; }
                      P(e)
```

[4] "Storage recycling" sounds a bit more trendy than "garbage collecting".

In each case, a variable (x) is declared, initialized (with the value e), and used (in the command `write` x). The difference is that, in the example on the left, declaration and initialization occur at the same time and, in the example on the right, declaration and initialization are separated.

The *principle of correspondence* asserts that, for any mechanism that binds a value to a name, there should be a corresponding mechanism that binds a value to the parameter of an abstraction. (The principle of correspondence was first formulated by Peter Landin (1966) and is discussed by Tennent (1977, 1981).)

In the example on the left, the value e is bound to the variable x in a declaration; in the example on the right, the value e is bound to the variable x by passing e as an argument to the procedure P.

We can use the principle of correspondence to reveal inconsistencies in Pascal, as shown in Figure 29. The first line refers to Pascal constant definitions of the form `const` $x = e$, for which there is no corresponding parameter mechanism. The second line refers to Pascal variable definitions of the form `var` $x : T$: there is no way of initializing such declarations, but there is a corresponding parameter mechanism — which does not have the keyword `var`. Finally, the third line refers to binding a memory address to a name: there is no declaration mechanism, but there is a parameter mechanism — that uses the keyword `var`!

Kind of binding	Block	Procedure
Value	constant definition	—
Memory contents	variable declaration	value parameter
Memory address	—	`var` parameter

Figure 29: Inconsistencies in Pascal

Thus Pascal is non-orthogonal with respect to declarations and parameter mechanisms (as are most other languages). Pascal lacks mechanisms for:

- ▷ passing constants to procedures; and
- ▷ binding addresses to names.

Actually, Pascal *does* provide a kind of address binding, in the form of the `with` statement. If f is a field of a record, in the context

```
with p↑ do
   begin
     .....
   end
```

f stands for $p{\uparrow}.f$. We can think of the `with` statement as providing an “anonymous binding” for the address $p{\uparrow}$. (Note that f can be used as an lvalue or an rvalue. Consequently, it is an address that is bound, not the contents of the address.)

Turing provides a `bind` statement with the specific function of binding address to names. A simple example is given below. Following the `bind` statement, and within the block containing it, the name M acts as a synonym for *Students* $[N]\uparrow.Marks$, just as a `var` parameter acts as a synonym for the corresponding argument in Pascal. The abbreviation is an aid both for someone reading the program (because it avoids the repetition of complex expressions) and for the compiler (because it suggests optimizations such as keeping the address M in a register). The `bind` statement is better than Pascal's `with` statement because it provides an explicit name for the address.

```
bind M to Students[N]↑.Marks
for i : Assignments
    M[3] := M[1] + M[2]
end for
```

A "complete" language would allow initialized declarations for constants, locations, and addresses, and would also provide corresponding mechanisms for passing parameters.

Exercise 46 Construct a table like Figure 29 for `C`. □

7.3 The Principle of Abstraction

An *abstraction mechanism* provides a means of giving a name to a syntactic unit and invoking that unit by using the name. In addition, it may be possible to parameterize the abstraction and provide arguments in the invocation.

Familiar examples of abstraction mechanisms include functions in `C` and procedures and functions in Pascal. We consider abstraction mechanisms in Pascal and use semantic concepts to suggest possible improvements.

7.3.1 Procedures

A simple view is that procedures abstract commands. This is not quite true, however, because procedures may contain local declarations. Procedures actually name syntactic units of the following form.

```
const
    constant definitions
type
    type definitions
var
    variable declarations
procedure and function declarations
begin
    statements
end
```

This syntactic unit is called a *block*. (Most implementations of Pascal allow the definitions and declarations in a block to be interleaved in an arbitrary order, subject only to the declaration-before-use rule.) A procedure declaration provides a name for the block, and allows variables to be passed to it by value or by reference.

Although procedures are the abstraction mechanism for blocks, blocks are not syntactic units of Pascal (although a complete Pascal program is a block). If Pascal was a true block-structured language, we could write

```
.....
begin
    block
end
.....
```

in any statement context. Algol68 provides this kind of block with full generality, and C provides a restricted form of block, with variable declarations only.

Block-structured programming emerged at a time when top-down program development was fashionable (Wirth 1971). Top-down development leads naturally to hierarchical, or nested, structures and provides a close match to block-structured programming. During the 70s, however, it became clear that nested blocks were inadequate for large-scale software development, and modules were introduced to provide better structuring facilities.

Although procedures can have parameters, the kinds of parameters they can have are severely limited. We cannot, for example, pass types as arguments. If we could, we could write

```
procedure Swap (T: type; x, y: T)
    var z: T;
    begin
        z := x; x := y; y := z
    end;
```

The template mechanism of C++ provides type parameters, with the restriction that all type-parameter processing must be done by the compiler.

7.3.2 Functions

Functions in Pascal are similar to procedures: the main difference is that they return a value and are invoked in an expression context rather than a statement context. Although functions abstract expressions, the body of a function is not an expression but a value-returning block. Again, the form of function bodies indicates a missing feature, because we cannot write a value-returning block in Pascal. If we could, it would be possible include statements such as the following in Pascal programs (the `return` statement compensates for Pascal's inadequate mechanism for returning values):

```
write (
    var i, s: integer;
    begin
        s := 0;
        for i := 1 to 10 do
            s := s + i;
        return s
    end
)
```

Functions, like procedures, accept only variables as arguments.

7.3.3 Types

Pascal provides type expressions that allow us to define arrays, records, pointers, sets, and files. It also provides abstraction over type expressions. There are, however, two anomalous features of the type abstraction mechanism. The first is the distinction between *name equivalence* and *structural equivalence*.

In mathematics, a definition of the form $x \triangleq E$ entitles us to use x and E interchangeably. Definition provides a means of *abbreviation*: we introduce x only because it is easier to write than E. Furthermore, if we write $y \triangleq E$, we are entitled to believe that $x = y$.

These assumptions do not hold for Pascal type definitions. If we write

```
type
    T1 = array [1..10] of integer;
    T2 = T1;
    T3 = array [1..10] of integer;
```

then we have $T_2 = T_1$ but not $T_3 \neq T_2$. This has several consequences, one of which is that we cannot write a procedure declaration such as

```
procedure P (a: array [1..10] of integer);
    .....
```

because we cannot provide an argument that matches the type of the parameter.

Standard Pascal uses *name equivalence* for types: the types T_1 and T_2 above are equivalent because their *names* are defined to be equivalent. If Pascal used *structural equivalence* (as Algol68 does), then T_1 and T_3 would be equivalent types. Since the problems of type equivalence were not realized until Pascal had been in use for some years, early Pascal compilers were often incompatible with one another in this respect (Welsh, Sneeringer, and Hoare 1977).

The second anomalous feature of type definitions is that we cannot parameterize them. It would be useful to define a type such as

```
type
    Matrix (M, N: integer) = array [1..M, 1..N] of real;
```

and then use this type in variable declarations such as

```
var
    M: Matrix (3, 4);
```

It would also be useful to have type parameters for types:

```
type
    SearchTree (KT, DT: type) =
      record
        Key: KT;
        Data: DT;
        left, right: ↑SearchTree
      end
.....
var
    S: SearchTree (integer, string);
```

Type parameters would enable us to define *dependent types*, in which the type of one field depends on the value of another field.

```
type
    String =
        record
           Len: integer;
           Txt: array [1..Len] of Char
        end
```

7.3.4 Selectors

Pascal has expressions for which there is no abstraction mechanism at all: namely, lvalues. We can write expressions such as $a[i]$, $r.f$, and $p\uparrow$ on the left side of an assignment statement, but we cannot name these expressions.

Tennent (1981) proposes the name "selector" for a mechanism that abstracts lvalues. Consider a simple Pascal data structure that represents a stack:

```
type
   Stack =
      record
         Sp: integer;
         Body: array [1..10] of real;
      end
```

Assume that we use the convention that the stack pointer points to the top component of the stack. The selector *Top* provides the lvalue of the top component.

```
selector Top (S: Stack);
   begin
      Top := S . Body [S . Sp]
   end;
```

We can use *Top* in exactly the same ways as we can use the expression it abbreviates, *S . Body* [*S . Sp*]. For example:

```
Top (S) := Top (S) + 1
```

Exercise 47 What is the difference (if any) between a selector and a function that returns a pointer? □

Exercise 48 The introduction of selectors into Pascal would create a problem that the designers of Pascal were careful to avoid. What is the problem?

Hint: this problem *does* exist in C and, in fact, is a common trap for beginners. □

Exercise 49 Write a brief (one to two page) critical review of (Ashcroft and Wadge 1982). Comment on what you consider to be its strengths and weaknesses and on its relationship to the course. □

Exercise 50 Pascal provides the type `set of` T with operations `in` (membership), `+` (union), `*` (intersection), `-` (difference), `<=` (subset), and `>=` (superset). C provides operators that work on bitstrings of limited length (typically 32 bits), including `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise exclusive OR), `<<` (shift left), and `>>` (shift right).

Compare these features of Pascal and C from denotational and operational viewpoints. □

Exercise 51 Suppose that the ability to define parameterized arrays was added to Pascal, as outlined in Section 7.3.2. Discuss the feasability of this addition, considering implementation issues and type checking. □

Exercise 52 A possible first step in designing a language using semantic principles is to define the semantic domains — the basic, denotable, expressible, and storable values. If you were designing a language, how would you set up the domains? You can make specific suggestions (such as "domain X should contain Y"), or you can suggest useful rules ("domain X should be a subset of domain Y"), or both. $\Box$

8 Syntax and Semantics

We have looked at various ways of defining a semantics for simple programming languages, but we have not investigated the meaning of the term "semantics" itself. In this section, we will review the concept of semantics as it is used in the mathematics of logic and functions.

Roughly, syntax is structure and semantics is meaning. We will illustrate this distinction with a simple and (hopefully) familiar example: propositional calculus.

8.1 Syntax and Semantics of Propositional Calculus

We can describe the propositional calculus as the language of *propositions*. A proposition is one of:

▷ a *propositional letter* P_0, P_1, ... (we will assume there are as many propositional letters as we need); or

▷ has one of the forms $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\neg\alpha)$, or $(\alpha \Rightarrow \beta)$, where α and β are propositions.

In addition to the rules for constructing compositions, we have three axioms (actually *axiom schemata*, since each axiom can be instantiated with many different propositions):

$$\begin{array}{l} (\alpha \Rightarrow (\beta \Rightarrow \alpha)) \\ ((\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma)) \\ (\neg\beta \Rightarrow \neg\alpha) \Rightarrow ((\neg\beta \Rightarrow \alpha) \Rightarrow \beta) \end{array}$$

and a rule of inference (*modus ponens*):

$$\frac{\alpha \qquad \alpha \Rightarrow \beta}{\beta}.$$

Although $\wedge$ and $\vee$ do not appear in the axioms or rules, we introduce them by definitions to shorten proofs:

$$\begin{array}{rcl} \alpha \wedge \beta & \triangleq & \neg(\alpha \Rightarrow \neg\beta) \\ \alpha \vee \beta & \triangleq & \neg\alpha \Rightarrow \beta. \end{array}$$

We use the axioms and inference rules to construct *proofs*. Let S be a set of propositions. Then α is *provable from* S if there is a sequence $\alpha_1, \alpha_2, \ldots, \alpha_n$ of propositions such that

1. $\alpha = \alpha_n$.
2. For $i \le n$, one of the following is true:

   ▷ α_i is an axiom;

   ▷ $\alpha_i \in S$; or

   ▷ α_i is the consequence of an inference rule in which the antecedents are drawn from $\alpha_1, \ldots, \alpha_{i-1}$.

Propositional calculus, presented in this way, is a purely syntactic theory. We can use it to construct proofs but, as Bertrand Russell pointed out:

Mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we say is true.

We can provide a semantics for propositional calculus in the following way. We introduce the set of *truth values* $\mathbb{T} = \{ \mathsf{T}, \mathsf{F} \}$ and we define an interpretation of the logical connectives by means of *truth tables*. Truth tables should be familiar: Figure 30 shows the truth table for the connective $\wedge$.

$\wedge$	T	F
T	T	F
F	F	F

Figure 30: Truth table for $\wedge$

Definition 44 *A* **truth assignment** *is a function from propositional letters to truth values.*

Definition 45 *A* **truth valuation** *is a function that assigns a truth value to every proposition that is consistent with the truth tables.*

For example, if $\mathcal{A}$ is a truth assignment and $\mathcal{V}$ is a truth valuation, and $\mathcal{V}(\alpha) = \mathcal{V}(\beta) = \mathsf{T}$, then necessarily $\mathcal{V}(\alpha \wedge \beta) = \mathsf{T}$.

Theorem 46 *For every truth assignment $\mathcal{A}$, there is a unique truth valuation $\mathcal{V}$ such that $\mathcal{V}(\alpha) = \mathcal{A}(\alpha)$ for all propositional letters α.*

Proof (Outline) The "exists" part is easy: we simply choose a truth valuation $\mathcal{V}$ such that $\mathcal{V}(\alpha) = \mathcal{A}(\alpha)$ for all propositional letters α. The main point of the theorem is that this valuation is unique: if $\mathcal{V}_1(\alpha) = \mathcal{V}_2(\alpha)$ for each propositional letter α, then $\mathcal{V}_1(\alpha) = \mathcal{V}_2(\alpha)$ for all propositions. We can prove this by induction over the syntax of propositions. □

Suppose that we have a set of propositions S and a proposition α, and we wish to show that α is a logical consequence of S. There are two ways of doing this:

1. We could construct a proof that S entails α.
2. We could show that, for every valuation $\mathcal{V}$, if $\mathcal{V}(\beta) = \mathsf{T}$ for every $\beta \in S$, then $\mathcal{V}(\alpha) = \mathsf{T}$.

We first consider whether the two methods are feasible. The second method is obviously feasible, although potentially slow, because the number of possible valuations is finite: it is in fact 2^n, where n is the number of propositional letters in S and α. Although it is not obvious that the first method is feasible, there are in fact algorithms that construct proofs (or disproofs) of arbitrary propositions.

The second, and more important, question is: do the two methods give the same results? The answer is, of course, yes.

Theorem 47 (Soundness) *If there is a proof of α, then $\mathcal{V}(\alpha) = \mathsf{T}$ for all truth valuations $\mathcal{V}$.*

Theorem 48 (Completeness) *If $\mathcal{V}(\alpha) = \mathsf{T}$ for all truth valuations $\mathcal{V}$, then there is a proof of α.*

Theorem 49 (Consistency) *If there is a valuation $\mathcal{V}$ such that $\mathcal{V}(\alpha) = \mathsf{F}$, then there does not exist a proof of α.*

There are several ways of proving the consistency of a theory; one of the most common is to construct a *model* for it. The system of truth assignments and valuations constitutes a model of the propositional calculus.

The main point of this discussion is that we can demonstrate the truth of a proposition either *syntactically* by providing a proof of it or *semantically* by trying all valuations (that is, constructing the truth table). The soundness and completeness theorems show that the syntactic and semantic methods are equivalent: which we use is purely a matter of convenience.

To put it another way: for propositional calculus, *truth and provability are equivalent*.

Although we tend to take them for granted, the soundness and completeness results are actually quite remarkable. They allow us to reason by mechanical symbol manipulation, without concern for the meaning of the symbols. If we follow the syntactic rules, the semantics will look after themselves. It is not a coincidence that these theoretical results, achieved early in the century, eventually led to mechanical (and then electronic) calculating devices.

8.2 Predicate Calculus

The predicate calculus extends the propositional calculus by adding the quantifiers $\forall$ and $\exists$. Like the propositional calculus, the predicate calculus is sound and complete: truth and provability are again equivalent.

Demonstrating the truth of a predicate is harder than demonstrating the truth of a proposition, however. Since quantifiers allow us to express facts about infinite sets, we cannot use a semantic method (such as truth tables) to establish truth, and so we are forced to rely on syntactic methods (that is, proofs). But the predicate calculus is undecidable, which means that there is no systematic way of constructing the proof (or disproof) of an arbitrary predicate. Consequently, a theorem-proving program for predicate calculus must be a *heuristic* program that looks for proofs in likely places but cannot be certain of finding one.

8.3 First-order Arithmetic

First-order arithmetic is the theory that we obtain by combining predicate calculus and natural numbers. It is also called "Peano arithmetic" although it was first described in modern form by Frege. We will assume in the following discussion that arithmetic is *consistent* (that is, there is no proof of the predicate F) because there are simple models of arithmetic.

In 1931, most logicians thought that truth and provability were equivalent throughout mathematics (and working mathematicians took this for granted, scarcely thinking about it). Gödel, in his famous "incompleteness theorem", showed that this was not so for arithmetic. The following is a brief overview of Gödel's reasoning.

Gödel realized that a theory that contained arithmetic could be used to make statements about itself. The first step is to encode formulas. We will consider formulas of one particular kind only: the formulas with one free variable that is a natural number. Examples of such formulas include $i > 0$ and $\exists j \,.\, i = 2 \times j$, and so on. We will use f to denote these formulas, and we will write $f[n]$ to denote the formula f with its free variable replaced by the natural number n.

The encoding is performed by a function $G_\#$. If f is a formula, the natural number $G_\#(f)$ is its *Gödel number*. It doesn't much matter how we define $G_\#$ provided that it has these two properties:

- ▷ If f and f' are distinct formulas, $G_\#(f) \neq G_\#(f')$.
- ▷ Given a Gödel number n, we can uniquely determine the formula f such that $G_\#(f) = n$, if such a formula exists.

For example, we could assign a natural number to each symbol of the language and encode formulas as follows: suppose that $ABC\ldots$ is a formula and that $a, b, c, \ldots$ are the numbers assigned to the symbols $A, B, C, \ldots$. Then $G_\#(ABC\ldots) = 2^a \cdot 3^b \cdot 5^c \cdots$. In 1931, encoding had never been used and Gödel's techniques were rather inefficient; nowadays, programmers perform this kind of encoding all the time (e.g. Assignment 2).

If f is a formula, we will write $P_r(f)$ to denote a proof of f, which is, of course, another formula. How can we say that f is provable? In a first-order theory, there is no way of saying "there exists a proof of f", but in arithmetic supplemented with Gödel numbers, we can write

$$\exists n \,.\, G_\#(P_r(f)) \quad = \quad n$$

which says "there is a number n which is the Gödel number of a proof of f", which is true if and only if there actually is a proof of f, that is, if f is provable. This trick enabled Gödel to formalize provability "within the system" — which is reasonable because, as we have seen, provability is a syntactic concept.

Gödel then introduced the following rather strange predicate:

$$B(n, x) \quad \equiv \quad \exists f \,.\, G_\#(f) = n \;\wedge\; G_\#(P_r(f[n])) = x$$

The formula $B(n, x)$ is true if n is the Gödel number of some formula f and x is the Gödel number of a proof of $f[n]$. There is a minor technical problem: in a first-order theory, we are not supposed to quantify over formulas. Gödel went to some lengths to show that $B(n, x)$ is definable using only first-order concepts.

Gödel's next step was to introduce the formula $\neg\exists x \,.\, B(n, x)$ which says "if n is the Gödel number of a formula f, then there is no proof of the formula $f[n]$". Since this is a formula with one free variable (n), we can give it a Gödel number: we define

$$m \quad = \quad G_\#(\neg\exists x \,.\, B(n, x)).$$

Kurt Gödel (1906–78) is best known for his incompleteness theorem, published in 1931 when he was 25. This paper included another important result: it is impossible to prove the consistency of a consistent system within the system. Gödel had previously proved the completeness of predicate calculus in his Ph.D. thesis, published in 1930. Gödel was born in Brno, studied and taught at the University of Vienna, emigrated to the U.S.A. in 1940, and worked at the Institute of Advanced Study (Princeton) until his death.

Finally, we consider the famous "Gödel sentence"

$$\mathcal{G} \quad = \quad \neg\exists x \,.\, B(m,x)$$

in which m is defined as above. $\mathcal{G}$ says: "if m is the Gödel number of a formula f, then $f[m]$ is unprovable." We know that m *is* the Gödel number of a formula (see above) and we can therefore write down the formula that is unprovable: it is $\neg\exists x \,.\, B(m,x)$. In other words, $\mathcal{G}$ is a formal statement that says "I am unprovable"!

It seems that there might be a paradox similar to the famous "This sentence is false" paradox, but in fact $\mathcal{G}$ is not a paradoxical statement. There are two possibilities:

- ▷ Suppose that $\mathcal{G}$ is false. Then, since $\mathcal{G}$ means "$\mathcal{G}$ is unprovable", we can infer that $\mathcal{G}$ *is* provable. But that would make first-order arithmetic inconsistent, since we would be able to prove a false statement.
- ▷ Suppose that $\mathcal{G}$ is true. Then, by its own assertion, we cannot prove it. Consequently, first-order arithmetic is incomplete, because it contains true statements that cannot be proved.

As stated above, there are good reasons for believing that arithmetic is consistent (and, of course, most of mathematics would collapse if it wasn't). Consequently, we take the alternative path, and deduce that $\mathcal{G}$ is true and arithmetic is incomplete.

The subtlety of Gödel's proof is the way in which it sits at the borderline between syntax and semantics. We can establish the non-provability of $\mathcal{G}$ syntactically ("within the system") but its truth is (and must be) a consequence of semantic reasoning. This subtlety has confused many great thinkers. For example, there are people who believe that Gödel's theorem demonstrates a fundamental difference between human and mechanical reasoning. The distinguished physicist and mathematician Roger Penrose, for example, has written two large books (1989, 1994) that purport to demonstrate mental processes are not subject to Gödelian limitations and that creative thinking must therefore depend on quantum fluctuations in our microtubules.

Gödel's incompleteness results were a devastating blow to Hilbert's program of formalizing all of mathematics. His results also converted much previous work (notably, the monumental *Principia Mathematica* of Russell and Whitehead) into mathematical curiosities. Most practicing mathematicians, however, have never shown much interest in foundational issues, and Gödel's theorem created no more than a ripple in the flood of new mathematics created in this century.

8.4 Functions

We can define a function f in either of two ways: we can give a formula that shows how to calculate $f(n)$ for particular values of n (intensional definition), or we can provide a set of pairs (extensional definition). For example, the intensional definition

$$f(n) \quad = \quad n^2 + 3$$

would be equivalent to the extensional definition

$$f \quad = \quad \{\,(0,3),\ (1,4),\ (2,7),\ (3,12), \ldots\,\}$$

if we could write down the entire set. (We could do this easily if f had a finite domain, but most interesting functions are defined on infinite domains, such as the natural numbers.)

From the foregoing discussion, it should be clear that the intensional definition is syntactic, and the extensional definition is semantic. Since we cannot actually write down extensional definitions of interesting functions, we always use the syntactic method of symbol manipulation to reason about functions.

Two functions are equal if they give the same value everywhere in their range. Formally, if $f : A \to B$ and $g : A \to B$, then

$$f = g \quad \Longleftrightarrow \quad \forall x \in A \, . \, f(x) = g(x).$$

There are usually many ways of defining a particular function intensionally. For example, the functions f_1 and f_2 defined by

$$\begin{aligned} f_1(n) &= n^2 - 1 \\ f_2(n) &= (n+1)(n-1) \end{aligned}$$

are equal extensionally although they are not equal intensionally.

Equality for functions with infinite domains is undecidable. We can, of course, prove particular equalities, such as $f_1 = f_2$, but there is no general algorithm for constructing such proofs.

8.5 Programming Languages

As we have seen, a denotational semantics gives the meaning of a program as a function. In view of the discussion above, can we really claim that this is actually "semantics", or are we merely pushing symbols around?

The answer depends on how we use the semantic equations. If we use them as a kind of clumsy interpreter, to compute a particular final state from a particular initial state, we are indeed "merely pushing symbols around". This was not the intention of the people who developed denotational semantics, in particular Strachey and Scott. Their view is that the denotational semantics associates with each program a function in the form of an infinite set of pairs of states. The purpose of Scott's theory of domains is to provide ways of reasoning about such functions, and thereby to provide a "model" for programming in the sense that truth tables are a "model" for propositional calculus.

9 Types

We have so far avoided any detailed account of types in our discussions of the semantics of programming languages. Our approach has in fact followed that of Strachey and Scott. Their view was that computers operate on uninterpreted bitstrings, and the "domains" of the theory (which typically contain values of all types) were an appropriate abstraction of these bitstrings.

Although it is true that processors operate on uninterpreted bitstrings, this is not a view shared by most programmers. A processor will add a bitstring representing a character to a bitstring representing a floating-point number without complaining, but the programmer will probably consider this operation to be a mistake. In this section, we investigate ways of ensuring that such mistakes do not occur.

Processors provide standard representations for characters, integers, and floating-point numbers, and they also provide operations on these values. This behaviour of processes appears in most programming languages as a *type discipline*. For example, we might write (in Pascal):

```
var m, n: integer;
var x, y: real;
begin
    m := n + 7;
    x := y + 3.0;
    x := m;
    n := y;
    .....
```

The compiler accepts the first three assignments and rejects the fourth. In the third assignment, it will insert code to convert the integer value of m to the corresponding floating-point value before assigning this value to x. The compiler, however, does not permit the implicit conversion from a floating-point value to an integer value required by the fourth assignment. This is a choice on the part of Pascal's designers, who felt that implicit conversions should never result in loss of information.

There is both a syntactic and a semantic aspect of types. The syntactic aspect can be seen above: we insert type declarations in the program, and the compiler uses them to decide whether the program is acceptable and to make the appropriate choice of operations (the first "+" above translates to integer addition and the second to real addition). The semantic aspect is that the program, so compiled, will indeed yield results of the expected type.

9.1 Syntax of Types

We start with a simple example: specifically, we will add a type system to the language L_1, for which we gave the denotational semantics in Section 2.6 on page 8. According to the semantics, the "meaning" of the expression $\texttt{true} + 3$ in the empty state $\langle\rangle$ is

$$\begin{aligned}\mathcal{V}[\![\texttt{true} + 3]\!]\langle\rangle &= \mathcal{V}[\![\texttt{true}]\!]\langle\rangle + \mathcal{V}[\![3]\!]\langle\rangle \\ &= \mathsf{T} + 3\end{aligned}$$

which is an undefined expression. The purpose of the type system is to prevent the occurrence of errors of this kind.

Definition 50 *A* **type error** *is said to occur when the evaluation of an expression yields a value of a type that is unacceptable in the context of the expression.*

For example, the evaluation of T in the example above yields a Boolean value, which is unacceptable in the context of integer addition. This definition and the following definitions are rather vague, but they will do for now.

Definition 51 *A program is* **well-typed** *if it satisfies a set of syntactic constraints intended to ensure that it does not contain potential type errors.*

Definition 52 *A* **type syntax** *is a set of rules that make it possible to check that a program is well-typed.*

Definition 53 *A* **type semantics** *is a theory that ensures that the execution of a well-typed program from any initial state cannot cause a type error.*

9.2 Type Rules for L_1

We use $\sigma, \tau, \ldots$ to denote types. If e is an expression then $e : \tau$ means "expression e has type τ". (Since we are considering only the syntax of types, the precise meaning of $e : \tau$ is immaterial, but it helps to have an intuitive idea of what is going on.)

Definition 54 *A* **type assumption** *A is a sequence $\langle x_1 : \tau_1, x_2 : \tau_2, \ldots \rangle$ in which the x_i are distinct variables and the τ_i are types.*

Definition 55 *A* **type judgment** *is a formula $A \vdash e : \tau$ with the meaning "under assumptions A, the expression e has type τ".*

If we can determine that e has type τ without any assumptions, we write simply $\vdash e : \tau$. The difference between $e : \tau$ and $\vdash e : \tau$ is similar to the difference between (the proposition) P and the statement $\vdash P$: the first is simply a proposition that may be true or false, and the second is a claim that the proposition has been proved true by an inference system.

For L_1 we use three types: `Nat` and `Bool` for expressions, and `Void` for commands. We cannot simply assume that the type of commands is `Void`: the statement c : `Void` implies that executing the command c will not fail because of a type error; thus all of the components of c must be well-typed. Figure 31 shows the type axioms and inference rules for L_1.

As an example, we use the rules of Figure 31 to show that the program

$x := 0$;
`while` $x \le 3$ `do` $x := x + 1$

is well-typed and has type `Void` under the assumption x : `Nat`. Figure 32 shows the proof with the same numbering convention that we used in Figure 11 on page 13.

$$\vdash n : \texttt{Nat}$$
$$\vdash \texttt{true} : \texttt{Bool}$$
$$\vdash \texttt{false} : \texttt{Bool}$$
$$\vdash \texttt{skip} : \texttt{Void}$$
$$x : \tau \vdash x : \tau$$

$$\frac{A \vdash e_1 : \texttt{Nat} \qquad A \vdash e_2 : \texttt{Nat}}{A \vdash e_1 + e_2 : \texttt{Nat}}$$

$$\frac{A \vdash e_1 : \texttt{Nat} \qquad A \vdash e_2 : \texttt{Nat}}{A \vdash e_1 \le e_2 : \texttt{Bool}}$$

$$\frac{A \vdash x : \tau \qquad A \vdash e : \tau}{A \vdash x := e : \texttt{Void}}$$

$$\frac{A \vdash c_1 : \texttt{Void} \qquad A \vdash c_2 : \texttt{Void}}{A \vdash c_1; c_2 : \texttt{Void}}$$

$$\frac{A \vdash b : \texttt{Bool} \qquad A \vdash c_1 : \texttt{Void} \qquad A \vdash c_2 : \texttt{Void}}{A \vdash \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \texttt{Void}}$$

$$\frac{A \vdash b : \texttt{Bool} \qquad A \vdash c : \texttt{Void}}{A \vdash \texttt{while}\ b\ \texttt{do}\ c : \texttt{Void}}$$

Figure 31: Type axioms and inference rules for L_1

1.1.1	$x : \texttt{Nat} \vdash x : \texttt{Nat}$
1.1.2	$\vdash 0 : \texttt{Nat}$
1.1	$x : \texttt{Nat} \vdash x := 0 : \texttt{Void}$
1.2.1.1	$x : \texttt{Nat} \vdash x : \texttt{Nat}$
1.2.1.2	$\vdash 3 : \texttt{Nat}$
1.2.1	$x : \texttt{Nat} \vdash x \le 3 : \texttt{Bool}$
1.2.2.1	$x : \texttt{Nat} \vdash x : \texttt{Nat}$
1.2.2.2.1	$x : \texttt{Nat} \vdash x : \texttt{Nat}$
1.2.2.2.2	$\vdash 1 : \texttt{Nat}$
1.2.2.2	$x : \texttt{Nat} \vdash x + 1 : \texttt{Nat}$
1.2.2	$x : \texttt{Nat} \vdash x := x + 1 : \texttt{Void}$
1.2	$x : \texttt{Nat} \vdash \texttt{while}\ x \le 3\ \texttt{do}\ x := x + 1 : \texttt{Void}$
1	$x : \texttt{Nat} \vdash x := 0; \texttt{while}\ x \le 3\ \texttt{do}\ x := x + 1 : \texttt{Void}$

Figure 32: Using the type inference rules

9.3 Semantics of Types

We would like to be certain that:

- ▷ if there is a set of assumptions A such that $A \vdash e : \texttt{Nat}$, then the evaluation of e yields a value in $\mathbb{N}$.
- ▷ if there is a set of assumptions A such that $A \vdash e : \texttt{Bool}$, then the evaluation of e yields a value in $\mathbb{T}$.
- ▷ if there is a set of assumptions A such that $A \vdash c : \texttt{Void}$, then the evaluation of c does not lead to any type errors.

We can formalize these requirements as follows:

$$\begin{aligned} A \vdash e : \texttt{Nat} &\;\Rightarrow\; \forall \sigma \,.\, \mathcal{V}[\![e]\!]\sigma \in \mathbb{N} \\ A \vdash e : \texttt{Bool} &\;\Rightarrow\; \forall \sigma \,.\, \mathcal{V}[\![e]\!]\sigma \in \mathbb{T} \\ A \vdash c : \texttt{Void} &\;\Rightarrow\; \forall \sigma \,.\, \mathcal{C}[\![c]\!]\sigma \in \mathbb{V} \end{aligned}$$

in which $\mathbb{V}$ is the singleton set containing the unique value *ok*. Since $\perp \notin \mathbb{V}$, the last statement asserts that well-typed commands do not fail.

To simplify the type correctness statements, we use a mapping from the syntactic types `Int`, `Bool`, and `Void` to the sets $\mathbb{N}$, $\mathbb{T}$, and $\mathbb{V}$: (Note that $\mathcal{D}$ is used in a different sense than its previous use as a semantic function for declarations.)

$$\begin{aligned} \mathcal{D}[\![\texttt{Nat}]\!] &= \mathbb{N} \\ \mathcal{D}[\![\texttt{Bool}]\!] &= \mathbb{T} \\ \mathcal{D}[\![\texttt{Void}]\!] &= \mathbb{V} \end{aligned}$$

These definitions provide a simple statement of the following theorems. We use x to denote an expression or a command and $\mathcal{M}$ to denote a general meaning function (previously $\mathcal{V}$ or $\mathcal{C}$).

Theorem 56 (Semantic Soundness) $A \vdash x : \tau \;\Rightarrow\; \forall \sigma \,.\, \mathcal{M}[\![x]\!]\sigma \in \mathcal{D}[\![\tau]\!]$.

We can prove Theorem 56 by induction over the syntax. Here are examples of typical cases. The induction hypothesis (IH) is the statement of the theorem itself.

Case `true`. From the type axioms,

$$\vdash \texttt{true} : \texttt{Bool}.$$

From the semantic equations,

$$\forall \sigma \,.\, \mathcal{M}[\![\texttt{true}]\!]\sigma = \mathsf{T}.$$

But $\mathsf{T} \in \mathbb{T}$ and $\mathbb{T} = \mathcal{D}[\![\texttt{Bool}]\!]$. Consequently,

$$\vdash \texttt{true} : \texttt{Bool} \;\Rightarrow\; \forall \sigma \,.\, \mathcal{M}[\![\texttt{true}]\!]\sigma \in \mathcal{D}[\![\texttt{Bool}]\!]$$

since both sides are true.

The cases `false` and n are similar.

Case $e_1 + e_2$**.** Assume $A \vdash e_1 + e_2 : \texttt{Nat}$. The only way that we could reach this conclusion using the inference rules is by first proving that

$$A \vdash e_1 : \texttt{Nat}$$

and

$$A \vdash e_2 : \texttt{Nat}.$$

By IH, it follows that

$$\forall \sigma \,.\, \mathcal{M}\,[\![\, e_1 \,]\!]\, \sigma \in \mathbb{N}$$

and

$$\forall \sigma \,.\, \mathcal{M}\,[\![\, e_2 \,]\!]\, \sigma \in \mathbb{N}$$

since $\mathcal{D}\,[\![\, \texttt{Nat} \,]\!] = \mathbb{N}$. By arithmetic,

$$\forall \sigma \,.\, \mathcal{M}\,[\![\, e_1 \,]\!]\, \sigma + \mathcal{M}\,[\![\, e_2 \,]\!]\, \sigma \in \mathbb{N},$$

and from the semantics it follows that

$$\forall \sigma \,.\, \mathcal{M}\,[\![\, e_1 + e_2 \,]\!]\, \sigma \in \mathbb{N}.$$

Consequently,

$$A \vdash e_1 + e_2 : \texttt{Nat} \quad \Rightarrow \quad \forall \sigma \,.\, \mathcal{M}\,[\![\, e_1 + e_2 \,]\!]\, \sigma \in \mathcal{D}\,[\![\, \texttt{Nat} \,]\!]\,.$$

The case $e_1 \leq e_2$ is similar.

The case `skip` : `Void` is similar to the case `true` : `Bool`.

Case $x := e$**.** This case is similar to $e_1 + e_2$, except that there are two cases two consider: either x and e are both of type `Bool` or they are both of type `Nat`.

The reasoning for the `if` and `while` commands is similar. The proof does not go through with the recursive rule for the `while` command, but it can be carried out (with suitable assumptions) using the fixpoint semantics of Section 4.

The converse of Theorem 56 is not true: there are programs in L_1 (and most other languages) that execute correctly but cannot be type-checked. Here is a simple example:

`if true then skip else` $n := 3 +$ `true`

It is particularly easy to construct a type system for the language L_1 because there is not much that can go wrong.

An unusual feature of L_1 is that there are no variable declarations. To use the type inference system, we must examine assignments and guess the appropriate types. For example, if the program contains the assignment $x := 7$, we "guess" that x : `Nat`. The type system will not allow inconsistent guesses.

Exercise 53 Show that it is not possible to find assumptions A such that

$$A \vdash (x := 7;\ x := \texttt{true}) : \texttt{Void}.$$

□

9.3.1 Classification of Type Checking

There are many ways in which a language processor can check types. These are three useful dimensions:

Strong *vs* **Weak** Strong type checking implies that a checked program cannot fail at run-time because of a type error. Weak type checking implies that a program is unlikely to fail at run-time because of a type error, but that type errors are possible.

Both C and Pascal are both weakly type checked in this respect, because their type checking systems have loopholes.

Static *vs* **Dynamic** Static type checking requires type checking to be completed before the program is executed. Dynamic type checking means that the types of operands are checked before the operations are applied at run-time.

Interpreted languages are usually checked dynamically. For example, LISP functions typically check the types of their operands before applying them.

Declared *vs* **Inferred** Some languages require the types of variables to be declared by the programmer, others do not require declarations.

C and Pascal require type declarations. LISP interpreters do not require declarations, but some LISP compilers do (or, at least, compile more efficient code if declarations are provided).

Common patterns include weak/static/declared, which describes C, Pascal, and a number of other compiled languages. Another common pattern is strong/dynamic/inferred, which includes LISP and Prolog.

Surprisingly, there are languages that combine the best of all features. SML, for example, can be described as strong/static/inferred: a compiled SML program cannot fail because of type errors; type checks are performed at compile-time; and programmers do not declare types.

9.3.2 Syntactic and Semantic Views of Types

Suppose that we perform type checking on a program and we find type errors. For example, if *succ* is defined on that natural numbers, what do we do about $succ\,(\pi)$?

One possibility is to say that a program such as $succ\,(\pi)$ is syntactically incorrect — in other words, it is not a program at all, and has the same status as a nonsensical expression such as $x+)y\times$. This approach simplifies the semantics of the language, because we can assume type correctness when we write the semantic equations. In fact, we implicitly adopted this approach in the discussion of L_1 and L_2.

Computers, however, do not behave in this way. Given an expressions such as $succ\,(\pi)$ (in the form of machine code), the processor will compute something, even though the result may be useless. The developers of denotational semantics wanted to model computation in this way.

Accordingly, in denotational semantics, all computations are performed on a domain D defined to be a solution of the recursive equation

$$D \;\cong\; \mathbb{T} + \mathbb{N} + \cdots + D \times D + D \to D + \cdots$$

The domain D contains basic values such as truth values and natural numbers, pairs (and hence tuples), all continuous functions on D, and perhaps some other stuff. It is an ordered domain in the sense of Section 3.6. Technically, it is called a *retract*, because it is a solution of a recursive domain equation and is shown to exist by means of fixed-point arguments. The symbol $\cong$ indicates that the two sides of the definition are isomorphic to one another, rather than being equal.

All functions in this model are defined as having type $D \to D$. We define additional types that are subsets of D with certain properties that are given below. Suppose that $A \subseteq D$ and $B \subseteq D$ are types. The statement $f : A \to B$ means

$$\forall x \in D \,.\, x \in A \Rightarrow f(x) \in B.$$

If $x \notin A$, we can say nothing about $f(x)$ except that $f(x) \in D$.

The properties that A must have in order to be a type are these.

- ▷ If $y \in A$, then all predecessors (in the domain ordering) of y must also belong to A. This property is called *downward closed* and it can be expressed formally as

$$\forall x, y \in D \,.\, x \sqsubseteq y \wedge y \in A \;\Rightarrow\; x \in A.$$

- ▷ If $x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_n$ is an ascending chain in A, then its least upper bound must also be an element of A. This property is called *directed complete* and it can be expressed formally as

$$x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_n \in A \quad \Rightarrow \quad \bigsqcup \{\, x_i \,\} \in A.$$

A set that is downward closed and directed complete is called an *ideal*. MacQueen *et al.* (1984) have shown that ideals can be used to construct very flexible type systems that even allow self-application ($f(f)$) to be given types.

9.4 Type Polymorphism

There are several mechanisms by which a single symbol can be given more than one type.

9.4.1 Overloading

A familiar mechanism, employed by many languages, is **overloading**. In C and Pascal, for example, the operator + (and other arithmetic operators) accepts arguments that are either integers or floats and returns values of the corresponding type. These languages provide a fixed number of operators that can be overloaded. More recent languages, such as Ada, allow programmers to define overloaded functions.

Overloading is syntactic, not semantic. We can consider + to be the name of two functions, *add-int* and *add-float*. When the compiler encounters an expression of the form $x + y$, it uses the type of the operands to determine which function to call.

9.4.2 Overloading with Dynamic Binding

Object oriented languages introduce a more flexible form of overloading. Messages have a syntax such as $o.m(a)$, in which o denotes an object, m a method (that is, a procedure or function), and a denotes an argument of the method.

The choice of m depends on the class (or type) of the object o. In the presence of inheritance, the compiler is not always able to determine the precise type of o. Consequently, the choice is left until run-time. Since the body of (or code for) the method is not bound to the name of the method until run-time, the mechanism is referred to as "dynamic binding". It is also a form of overloading since a single name (m) may denote several methods.

9.4.3 Parametric Polymorphism

Strachey introduced the terms "*ad hoc* polymorphism" for overloading and "parametric polymorphism" for the more general kind of polymorphism discussed in this section.

Milner introduced two important ideas in a single paper (1978): the first was polymorphism and the second was the inference of types in the absence if type declarations.

Languages such as LISP and Prolog acquire much of their expressiveness from the polymorphic nature of their built-in functions. In LISP, for example, $hd(x)$ returns the first component of the list x, whatever the type of the components of x. (LISP functions actually have peculiar names such as `car` and `cdr`. In this section, we use the names that we have used previously for list manipulation functions.) LISP systems incur a small performance cost for this flexibility, because the interpreter must check that the argument of *hd* is a list cell.

Milner demonstrated the possibility of combining flexibility and efficiency. In the type system of ML[5] the type of a general list is α^* (read "list of αs") in which α is a *type variable*. (The type α^* is actually written as `'a list` in SML.) The type of *hd* is $\alpha^* \to \alpha$: given a list of αs, *hd* returns a value of type α.

Consider the following definition of a function that operates on lists.

$$map\,(f, x) \quad = \quad \texttt{if}\ null\,(x)\ \texttt{then}\ nil\ \texttt{else}\ cons\,(f(hd(x)), map\,(f, tl(x))).$$

Although there are no type declarations in this definition, we can infer the type of each variable.

▷ To start with, the standard functions have types defined in terms of type variables (note that these types are patterns — α does not necessarily denote the same type in each case):

$$\begin{aligned} nil &: \alpha^* \\ hd &: \alpha^* \to \alpha \\ tl &: \alpha^* \to \alpha^* \\ cons &: \alpha \times \alpha^* \to \alpha^* \\ null &: \alpha^* \to Bool \end{aligned}$$

[5]ML stands for *MetaLanguage*. The current version of ML is *Standard ML*, or SML.

- ▷ We can see that x must be a list, because it is used as the argument of *hd* and *tl*. Suppose $x : \alpha^*$. Then $hd(x) : \alpha$ and $tl(x) : \alpha^*$.
- ▷ We can also see that f must be a function, since it is applied to the argument $hd(x)$. Suppose that $f : \alpha \to \beta$.
- ▷ It follows that $f(hd(x)) : \beta$. Since $f(hd(x))$ is the first argument of *cons*, the second argument of *cons* must have type β^*. Thus $map\,(f, tl(x)) : \beta^*$.
- ▷ Consequently, $Map : (\alpha \to \beta) \times \alpha^* \to \beta^*$.

This is the *most general type* of *Map*. It says that *map* takes a list of one type and, using a conversion function, returns a list of another type. For example, if f is $even : Nat \to Bool$ then, in the application $map\,(even, x)$, the function *map* has the *particular* type $(Nat \to Bool) \times Nat^* \to Bool^*$.

We do not require that *map* has the same type each time it occurs in the program. If we assume that

$$\begin{aligned} toks &: String^* \\ len &: String \to Nat \\ sqr &: Nat \to Nat \end{aligned}$$

then $map\,(sqr, map\,(len, toks))$ returns a list of the squares of the lengths of the tokens in the list *toks*. The two occurrences of *map* have types

$$(Nat \to Nat) \times Nat^* \to Nat^*$$

and

$$(String \to Nat) \times String^* \to Nat^*.$$

Milner gave an algorithm for type inference based on *unification*. The algorithm starts by assigning distinct type variables to all symbols of unknown types. It then sets up a set of constraints that govern the choice of values for the variables, using reasoning similar to that above. Finally, the algorithm uses unification to determine the most general types satisfying the equations.

A *unifier* is a set of substitutions that make two expressions equivalent. The expressions may contain constants and, unless they are identical, they must also contain variables. For example, the type expressions

$$\alpha^* \to Nat$$

and

$$String^* \to \beta$$

are unified by the substitution

$$u = [\alpha = String, \beta = Nat].$$

There is an algorithm, first given by Robinson (1965), that finds the most general unifier of any two expressions, or reports failure if the expressions cannot be unified.

9.5 The Curry-Howard Isomorphism

In this section, we consider type rules for various additional constructs, and an interesting relationship first noticed by Haskell Curry. We focus on the types of expressions, rather than commands, and we consider the types of abstractions and applications, following the simple patterns of λ-calculus.

We will use a *syntax of types*. There are some basic types, such as *Nat* and *Bool*, that will not be of much concern. If σ and τ are types, we assume the existence of the following types:

$\sigma \to \tau$ is the type of functions from σ to τ.

$\sigma \times \tau$ is the type of pairs (records) $\langle M, N\rangle$ in which $M : \sigma$ and $N : \tau$.

$\sigma + \tau$ is the type of unions (variant records) which contain an element that may be of type σ or τ.

Consider abstraction first. The type inference rule is

$$\frac{\begin{array}{c}[x : \sigma]\\ M : \tau\end{array}}{\lambda x \,.\, M : \sigma \to \tau}.$$

This is read: "if, under the assumption that x has type σ, we can show that M has type τ, then we can infer that the function $\lambda x \,.\, M$ has type $\sigma \to \tau$". The square brackets around $x : \sigma$ indicate that this is an assumption that can be *discharged*.

Application is simpler: if we have a function of type $\sigma \to \tau$, we must apply it to an argument of type σ, yielding a result of type τ.

$$\frac{M : \sigma \to \tau \qquad N : \sigma}{MN : \tau}$$

Next, we consider pairs. In order to construct a pair $\langle M, N\rangle$, we need expressions of appropriate types.

$$\frac{M : \sigma \qquad N : \tau}{\langle M, N\rangle : \sigma \times \tau}$$

If we have a pair, we use the *projection functions* $\pi_1 : \sigma \times \tau \to \sigma$ and $\pi_2 : \sigma \times \tau \to \tau$ to extract a component from it.

$$\frac{M : \sigma \times \tau}{\pi_1\,(M) : \sigma} \qquad\qquad \frac{M : \sigma \times \tau}{\pi_2\,(M) : \tau}$$

For unions, we need *injection functions*. Given an expression of type σ, we use one of the injections $\iota_1 : \sigma \to \sigma + \tau$ or $\iota_2 : \tau \to \sigma + \tau$ to create a union value of type $\sigma + \tau$:

$$\frac{M : \sigma}{\iota_1\,(M) : \sigma + \tau} \qquad\qquad \frac{N : \tau}{\iota_2\,(N) : \sigma + \tau}$$

To extract values from unions, we need a way of recognizing the type of the component. Functional languages typically use `case` expressions to accomplish this. The underlying mechanism (tagging) is hidden from the programmer.

$$\frac{M : \sigma + \tau \qquad f : \sigma \to \upsilon \qquad g : \tau \to \upsilon}{\texttt{case}\ M = \iota_1\,(x) \to f(x) \mid M = \iota_2\,(y) \to g(y)\ \texttt{end} : \upsilon}$$

The type inference rules have an interesting property which we can see by performing the following transformation. Remove all the terms, leaving only the types; change the type operators $\to$, $\times$, and $+$ to the logical connectives $\Rightarrow$, $\wedge$, and $\vee$. Then the transformed type inference rules are the natural deduction rules for propositional calculus. This correspondence (which can be made more precise) is called the *Curry-Howard isomorphism*.

The Curry-Howard isomorphism has several important consequences, which we only touch upon here (see (Girard 1989) for details). The first is that it tells us that type inference is always possible. (That is, given a well-typed program without type declarations, we can determine the types of all the variables.) This follows from the fact that, since there is a correspondence between inference rules, there must be a similar correspondence between proofs. Since there is a proof of every valid proposition, there must also be a deduction of every correct typing.

The isomorphism also suggests some questions. For example, are there type inference rules that correspond to the natural deduction rules for quantifiers? There are several ways of introducing quantified types, and we will consider the ways that are currently considered to be most useful.

- ▷ The type $\forall\alpha\,.\,T$, in which T is a type expression, is a *universal type*. (We will use $\alpha, \beta, \ldots$ for type variables.) For example, the identity function $\lambda x\,.\,x$ is considered to have type $\forall\alpha\,.\,\alpha \to \alpha$ since, given an argument of any type, it returns a result of the same type. Universal types correspond roughly to Milner's polymorphic types, discussed above.
- ▷ The type $\exists\alpha\,.\,T$ is an *existential type*. Existential types are used to reason about abstract data types. We know that α is a type, but all our information about it is confined to T. Typically T will be a *signature*, giving the types of the operations provided by the abstract data type. The existential quantification "protects" the type in the sense that the type checking mechanism allows us to call functions exported by the abstract data type but not to use the type in any other way. (For more details, see (Mitchell and Plotkin 1985).)

10 A Semantic Model for Object Oriented Programming

Discussions about object oriented programming are often hindered by disagreement about basic concepts. Unlike functional programming, grounded in λ-calculus, or logic programming, grounded in logic, object oriented programming lacks a simple model that we can use as a basis for definition and discussion.

There is, of course, a large and useful body of work in which the standard techniques of semantics, based on higher order typed λ-calculus, are used to explain object oriented programming, often with emphasis on type-correctness. But there remains a lingering suspicion that these techniques, despite their power, somehow miss the point. Since object oriented programming seems to be simple and appealing to programmers, perhaps there should be a simple model that accurately describes its salient features.

But what are the salient features of object oriented progamming? They include at least object identity, local state, and dynamic binding. It is not so obvious that a model of computation should describe inheritance, because inheritance is primarily a compile-time issue. A model should be able to describe delegation, however, since delegation decisions are made at run-time.

The conventional approach to semantics is to describe simple types—booleans, integers, and so on—first, then introduce additional machinery for product, unions, and recursive types. The approach uses a state that is more complicated than a simple environment, but which does not need to be extended for additional data structures.

The model should also describe the more controversial aspects of objects, such as side-effects, aliasing, the complicated forms of recursion which arise from inheritance, the use of *self* to denote the current object, and cyclic data structures. Descriptions of object oriented programming based on standard semantics need complex mechanisms to handle these phenomena, if they can handle them at all. The model described here handles all of them in a natural and simple way—which is not to say that they thereby cease to be problematic!

10.1 Object Oriented Programming

We characterize object oriented programming in the following way. An *object* has local state and the ability to perform certain actions. The local state is defined by the values of the *instance variables* of the object. The value of each instance variable is another object. Each action is called a *method*. An object may send a message to another object, requesting it to perform one of its actions: the message contains the name of a method and possibly some arguments. The binding between the name and the body of a method is created (conceptually, at least) when the object receives a message: this is called *dynamic binding*.

We are concerned primarily with systems in which each object is an instance of a particular *class*. All instances of a class respond to the same set of messages in the same way. A class defines the effect of each acceptable message either explicitly or by *inheriting* from another class.

It is natural to model an object oriented computation as a directed graph in which each vertex represents an object and each edge represents a link between objects. In our model, edges are labeled with the names of instance variables.

Within this framework, we can model simple values, records, and recursive data stuctures such as lists, trees, and graphs. A record with n fields is represented by a vertex with n out-edges. We model dynamic binding by associating methods with vertices.

10.2 The Graph Model

The formal basis for the graph model is a collection of sets, shown in Figure 33. The middle column shows the names of the sets, with a brief description on the left and some typical members on the right.

Description	Name of set	Typical members
Class names	$\mathcal{C}$	$\mathit{Root}, \alpha, \beta, \gamma$
Variable names	$\mathcal{L}$	x
Method names	$\mathcal{N}$	m, add
Programs	$\mathcal{P}$	P, Q, R
Vertices	$\mathcal{V}$	a, c, r, u, v, w

Figure 33: Sets used in the model

A state consists of a directed graph along with three of its vertices. A vertex has two components: a class name and an index chosen from the integers. We define the selector "class" to extract the class from a vertex:

$$\text{class}\,((\alpha, i)) \quad \triangleq \quad \alpha.$$

The index is used only to distinguish instances of the class. For example, the vertices corresponding to instances of class α are $(\alpha, 0)$, $(\alpha, 1)$, $\ldots$.

The partial function E defines the edges of the graph. If there is an edge from u to v with label x, then $E(u, x) = v$. In this case, we may also write $(u, x, v) \in E$. Otherwise, E is undefined. We use the symbol "†" to define modified edge functions. If $E' = E \dagger (v, y, w)$ then

$$E'(u, x) \quad = \quad \begin{cases} w, & \text{if } u = v \text{ and } x = y, \\ E(u, x), & \text{otherwise.} \end{cases}$$

A *state* is a tuple (V, E, c, a, r) in which

$$\left.\begin{array}{ll} V \subseteq \mathcal{V} & \text{is a finite set of vertices,} \\ E\colon V \times \mathcal{L} \rightharpoonup V & \text{is the } \textit{edge} \text{ function,} \\ c \in V & \text{is the } \textit{current} \text{ vertex,} \\ a \in V & \text{is the } \textit{argument} \text{ vertex, and} \\ r \in V & \text{is the } \textit{result} \text{ vertex.} \end{array}\right\} (*)$$

We require the set of vertices to be finite for two reasons. First, only a finite number of vertices can be created by a finite computation. Second, no technical problems are involved in choosing an index for a new vertex.

Notation

- The set of states (Σ) consists of all tuples $\sigma = (V, E, c, a, r)$ satisfying the conditions (*) above, and the "bottom" state, $\perp$.
- Subscripts applied to σ apply implicitly to all components of the state:
$$\sigma_i \quad \equiv \quad (V_i, E_i, c_i, a_i, r_i).$$
- We write $\sigma.r$ to abbreviate "r, where $\sigma = (V, E, c, a, r)$", and similarly for the other components of the state.
- If σ is a state and v is a vertex, then $\sigma \triangleright v$ is the same state as σ, but with result vertex v. The vertex set of the new state is assumed to contain v. Formally:
$$(V, E, c, a, r) \triangleright v \quad \equiv \quad (V \cup \{ v \}, E, c, a, v).$$

10.3 Abstract Syntax

The set of programs $\mathcal{P}$ consists of all programs that can be generated inductively using the abstract syntax rules of Figure 34. The last two programs in Figure 34 are constructed from simpler programs; we refer to them as *compound programs*.

$$P \quad \longrightarrow \quad \texttt{skip} \mid \texttt{self} \mid \texttt{arg} \mid \texttt{new}\ \alpha \mid x \mid \texttt{store}\ x \mid P; P \mid m(P)$$

Figure 34: Abstract syntax of simple and compound programs

10.4 Operational Semantics

We provide an operational semantics for programs by defining the effect that each program has on the state. Formally, the operational semantics gives evaluation rules for programs in the form
$$\mathcal{M}\,[\![P]\!]\,\sigma_0 \quad = \quad \sigma_1$$
which means "the result of evaluating the program P in the state σ_0 is a new state σ_1".

10.4.1 Simple Programs

Figure 35 defines the meanings of the various kinds of program. The program `skip` is the identity function. The next four programs are expressions: they yield a state σ in which the result vertex $\sigma.r$ has a particular value.

The program `new` α introduces a new vertex into the graph. The new vertex is (α, i), in which i is a unique index for the class α. The function next, defined by
$$\text{next}(\alpha, V) \quad \triangleq \quad \begin{cases} 0 & \text{if } \{ j \mid (\alpha, j) \in V \} = \emptyset, \\ 1 + \max \{ j \mid (\alpha, j) \in V \} & \text{otherwise.} \end{cases}$$
chooses an index that is zero for an unpopulated class and larger than the maximum index of a populated class.

$$\begin{array}{rcl}
\mathcal{M}[\![\,\texttt{skip}\,]\!]\sigma & = & \sigma \\
\mathcal{M}[\![\,\texttt{self}\,]\!]\sigma & = & \sigma \triangleright c \\
\mathcal{M}[\![\texttt{arg}\,]\!]\sigma & = & \sigma \triangleright a \\
\mathcal{M}[\![\texttt{new}\ \alpha\,]\!]\sigma & = & \sigma \triangleright (\alpha, \mathrm{next}(\alpha, V)) \\
\mathcal{M}[\![x\ ]\!]\,\sigma & = & \sigma \triangleright E(c, x) \\
\mathcal{M}[\![\texttt{store}\ \ x\,]\!]\sigma & = & (V, E \dagger (c, x, r), c, a, r) \\
\mathcal{M}[\![P\ ; Q]\!]\,\sigma & = & \mathcal{M}[\![Q\ \ ]\!]\,(\mathcal{M}[\![P\ \ ]\!]\sigma\ ) \\
\mathcal{M}[\![\,m(P)\,]\!]\,\sigma_0 & = & (V_2, E_2, c_0, a_0, r_2) \\
& & \quad \texttt{where}\ \ \mathcal{M}[\![\,P\,]\!]\,\sigma_0 = \sigma_1 \\
& & \quad \texttt{and}\ \ \ \ \ \mathcal{M}[\![\,\Phi(\mathrm{class}\,(r_0)\,, m)\,]\!]\,(V_1, E_1, r_0, r_1, r_1) = (V_2, E_2, c_2, a_2, r_2)
\end{array}$$

Figure 35: Operational semantics of simple programs

10.4.2 Compound Programs

The program $m(P)$ "sends a message" m with argument P to the result vertex, r_0. The operational semantics uses a partial function $\Phi : \mathcal{C} \times \mathcal{N} \to \mathcal{P}$ to obtain the program corresponding to the method name m. Informally, Φ takes a class name and a method name as arguments and returns a program. This corresponds to "dynamic binding".

We can interpret the evaluation of $m(P)$ as follows.

1. The initial state is $(V_0, E_0, c_0, a_0, r_0)$, in which r_0 is the "receiver" of the message.
2. The argument P is evaluated in the initial state, yielding a new state $(V_1, E_1, c_1, a_1, r_1)$, in which r_1 is the value of the argument.
3. The class of the receiver is $\alpha = \mathrm{class}\,(r_0)$.
4. The method invoked is $\Phi(\alpha, m)$. This method is evaluated in the state $(V_1, E_1, r_0, r_1, r_1)$, in which the current object, r_0, is the receiver and the argument, r_1, is the value of P. Evaluating the body yields the new state $(V_2, E_2, c_2, a_2, r_2)$.
5. The final state is $(V_2, E_2, c_0, a_0, r_2)$, in which the values of the current object and the argument have been restored from the initial state.

The inference rule for $m(P)$ requires that the current vertex, c, and the argument vertex, a, be saved and restored, but does not provide a mechanism for doing this. We could incorporate a mechanism for saving and restoring into the semantics, using, for example, a linked list of saved (c, a) pairs to represent a stack. Such a mechanism would introduce complexity into the semantics but would not have any apparent advantages.

10.4.3 Validity Conditions

Programs may fail in various ways. The failure of program P in state σ is described by the rule $\langle P, \sigma \rangle \to \bot$. The following failures are possible.

1. The program x will fail in state (V, E, c, a, r) if $E(c, x)$ is undefined.
2. The program $m(P)$ will fail in state σ_0 if either
   (a) (argument failure) $\mathcal{M}[\![P]\!]\sigma_0 = \bot$, or
   (b) (method failure) $\mathcal{M}[\![P]\!]\sigma_0 = \sigma_1$ and
   $$\mathcal{M}[\![\Phi(\text{class}(r_0), m)]\!](V_1, E_1, r_0, r_1, r_1) \quad = \quad \bot.$$
3. A program that contains direct or indirect recursive messages may fail to terminate.

The operational semantics does not allow a final state to be inferred for a program that fails to terminate.

10.4.4 Properties of the Operational Semantics

The model has a number of useful properties, which are summarized in the following lemma.

Lemma 57 *Let P be a program and let $\langle P, (V_0, E_0, c_0, a_0, r_0)\rangle = (V_1, E_1, c_1, a_1, r_1)$. Then:*

1. *$V_1 \supseteq V_0$ and $c_1 = c_0$ and $a_1 = a_0$.*
2. *If P does not contain* `store`*, then $E_1 = E_0$.*

The proof of the lemma is a straightforward induction over the program semantics given in Figure 35. It follows from the lemma that, in the rule for evaluating a method invocation, $c_1 = c_0$, $a_1 = a_0$, $c_2 = r_0$, and $a_2 = r_1$. Consequently, we can write this rule in the simpler form

$$\begin{aligned}\mathcal{M}[\![m\ (P)]\!]\sigma_0 \quad &= \quad (V_2, E_2, c_0, a_0, r_2)\\ &\quad\quad \texttt{where}\quad \mathcal{M}[\![P]\!]\sigma_0 = (V_1, E_1, c_0, a_0, r_1)\\ &\quad\quad \texttt{and}\quad\ \ \mathcal{M}[\![\Phi(\text{class}(r_0), m)]\!](V_1, E_1, r_0, r_1, r_1) = (V_2, E_2, c_0, a_2, r_2).\end{aligned}$$

The model provides encapsulation. The value of an instance variable x can be accessed (by the program x) or changed (by the program `store` x) only when its owner is the current object. Variables can be accessed from outside classes only if appropriate methods are provided, as in Smalltalk.

10.5 Examples

In the formal language, all methods require exactly one argument. Many of the methods introduced in this section are unary functions that do not access their argument. Other methods are binary functions that do access their arguments. We adopt the convention that, in a function definition, the name of a binary function is followed by "()" to indicate that an argument is expected.

10.5.1 Booleans

A Boolean object is an instance of either class *True* or class *False*. Each class provides three methods, *not*, *and*, and *or*, with different implementations. Figure 36 shows these classes. The table provides elements of the function Φ. The formal expression of Figure 36, for example, would be

$$\begin{aligned}\Phi \;=\; \{ \;& (\mathit{True}, \mathit{not}, (\texttt{new}\ \mathit{False})), (\mathit{True}, \mathit{and}, (\texttt{arg})), (\mathit{True}, \mathit{or}, (\texttt{new}\ \mathit{True})), \ldots, \\ & (\mathit{False}, \mathit{not}, (\texttt{new}\ \mathit{True})), (\mathit{False}, \mathit{and}, (\texttt{new}\ \mathit{False})), (\mathit{False}, \mathit{or}, (\texttt{arg})), \ldots \;\}\end{aligned}$$

Method	`class` *True*	`class` *False*
not	`new` *False*	`new` *True*
and()	`arg`	`self`
or()	`self`	`arg`
implies()	`arg`	`new` *True*
iff()	`arg`	`arg`; *not*

Figure 36: The Classes *True* and *False*

10.5.2 Natural Numbers

Instances of the class *Zero* represent the number 0; instances of the class *Pos* represent numbers greater than zero. Figure 37 shows these classes. The blank entries in class *Zero* indicate that this class does not provide the methods *link* and *pred*.

Method	`class` *Zero*	`class` *Pos*
iszero	`new` *True*	`new` *False*
succ	`new` *Pos*; `store` *x*; *link*(`self`); *x*	`new` *Pos*; `store` *x*; *link*(`self`); *x*
link()		`arg`; `store` *next*
pred		*next*
add()	`arg`	*pred*; *add*(`arg`); *succ*
eq()	`arg`; *iszero*	`arg`; *eqr*(`self`)
eqr()	`new`; *False*	`arg`; *pred*; *eq*(`self`; *Pred*)
lt()	`arg`; *iszero*; *not*	`arg`; *ltr*(`self`)
ltr()	`new` *False*	`arg`; *pred*; *lt*(`self`; *pred*)

Figure 37: The Classes *Zero* and *Pos*

The natural number $n > 0$ is represented by an instance u_1 of class *Pos*, where u_1 is the root of a linked list containing $n - 1$ edges. In other words, the state graph contains the edges

$$(u_1, \mathit{next}, u_2), \ldots, (u_{n-1}, \mathit{next}, u_n).$$

The call $v; \mathit{succ}()$, in which v is a vertex representing the natural number n, constructs a vertex representing $n + 1$ by creating a new node and linking it to n via an edge *next*, using the auxiliary method *link*. The edge label *next* must be the same for all numbers, but is not a reserved name — it can be freely used for other purposes.

The call $v; pred()$, in which v is a vertex representing the natural number n, returns the tail of the list rooted at v, which represents the predecessor of n.

Some functions, such as *eq* (equal) and *lt* (less than) require auxiliary functions (*eqr* and *ltr*). The auxiliary functions perform a similar operation to the primary functions, but with the argument and receiver interchanged. This corresponds to "double dispatching".

A consequence of the use of double dispatching is that the classes *True* and *False* do not play an important role in the implementation of natural numbers. Only the predicates *iszero*, *eq*, and *lt* return instances of *True* and *False*. This is because we have taken to an extreme the idea that object oriented programs should handle case analysis by dynamic binding.

10.5.3 An Example Program

We adopt the convention that a "main program" is started by invoking the method *entry* in the class *Main*. Here is a simple program:

```
class Main
   method entry
      new Zero; succ; store one;
      succ; add(self; one)
```

This program computes the successor of zero, stores it as "*one*", computes the successor of the same number, and adds "*one*". The result is a data structure that represents the natural number 3. Figures 38 and 39 show the computation in tabular form.

The first column shows the statement executed. The next three columns show the effect of the statement on the special vertices c, a, and r. The fifth column shows new edges added to the graph. The columns contain only entries that are relevant to the computation; a blank indicates that nothing has happened. The numbers in the sixth column correspond to the notes below.

1. The program is implicitly initialized by creating the vertex $(Root, 1)$, abbreviated to r_1 in the table. In similar fashion, z_i stands for $(Zero, i)$, and p_i stands for (Pos, i).

2. The column headed E does not show the entire edge relation, but only the edge most recently added to the state graph. Although in general a new edge may override an old edge, this does not happen in the example.

3. The program `ret` is not part of the formal language. It is used in the table to show the restored context after a method has returned. Although both the current object (c) and the argument (a) are restored, the table shows only the current object.

4. The method *add* is invoked three times, illustrating recursion. The recursion terminates when the receiver of *add* is an instance of *Zero*.

5. When the program terminates, the current vertex is r_1, as it was initially. The result is p_4, which corresponds to the natural number 3, since

$$\{ (p_4, next, p_3), (p_3, next, p_1), (p_1, next, z1) \} \subseteq E.$$

Program	c	a	r	E	Note
	r_1				1
`new` *Zero*			z_1		
succc	z_1				
`new` *Pos*			p_1		
`store` x				(z_1, x, p_1)	2
link(`self`)	p_1	z_1			
`arg`			z_1		
`store` *next*				$(p_1, next, z_1)$	
`ret`	z_1				3
x			p_1		
`ret`	r_1				
`store` *one*				(r_1, one, p_1)	
succ	p_1				
`new` *Pos*			p_2		
`store` x				(p_1, x, p_2)	
link(`self`)	p_2	p_1			
`arg`			p_1		
`store` *next*				$(p_2, next, p_1)$	
`ret`	p_1				
x			p_2		
`ret`	r_1				

Figure 38: Execution of program: first part

The example shows a rather "functional" style of programming. To see the difference between our model and a purely functional model, consider adding the following method to class *Pos*:

```
method infinity
   new Pos; store x; link(x)
```

Writing ∞ to stand for this object, and using mathematical notation, the model can execute the following programs in bounded time (n stands for any finite instance of *Pos*):

$$\begin{aligned} succ(\infty) &\rightarrow \infty \\ pred(\infty) &\rightarrow \infty \\ n < \infty &\rightarrow True \\ n + \infty &\rightarrow \infty \end{aligned}$$

The following computations, however, do not terminate:

$$\begin{gathered} \infty + n \\ \infty + \infty \\ \infty = \infty \end{gathered}$$

Program	c	a	r	E	Note
add(`self`; one)	p_2	p_1			4
`self`			p_2		
$pred$	p_2				
$next$			p_1		
`ret`	p_2				
add(`arg`)	p_1	p_1			
`self`			p_1		
$pred$	p_1				
$next$			z_1		
`ret`	p_1				
add(`arg`)	z_1	p_1			
`arg`			p_1		
`ret`	p_1				
$succ$	p_1				
`new` Pos			p_3		
`store` x				(p_1, x, p_3)	
$link$(`self`)	p_3	p_1			
`arg`			p_1		
`store` $next$				$(p_3, next, p_1)$	
`ret`	p_1				
x			p_3		
`ret`	p_1				
`ret`	p_2				
$succ$	p_3				
`new` Pos			p_4		
`store` x				(p_3, x, p_4)	
$link$(`self`)	p_4	p_3			
`arg`			p_3		
`store` $next$				$(p_4, next, p_3)$	
`ret`	p_3				
x			p_4		
`ret`	p_2				
`ret`	r_1				5

Figure 39: Execution of program: second part

10.6 Completeness

We can show that the graph model is complete by demonstrating that, corresponding to an arbitrary Turing machine, there is a graph model (GM) program that can perform an equivalent computation.

Theorem 58 *GM is computationally complete.*

Proof Let $T = (K, \Sigma, \delta, s)$ be a Turing machine with

- ▷ states, K, where $h \notin K$;
- ▷ alphabet, Σ, where $L, R \notin \Sigma$;
- ▷ transition function $\delta : K \times \Sigma \to (K \cup \{ h \}) \times (\Sigma \cup \{ L, R \})$;
- ▷ initial state, $s \in K$.

We construct a GM program that simulates T — see Figure 40. Assume that $\delta(q, \sigma) = (q', \sigma')$ and define $\delta_K(q, \sigma) = q'$ and $\delta_\Sigma(q, \sigma) = \sigma'$. For each $\sigma_i \in \Sigma \cup \{ L, R \}$, define a class σ_i. For each $q_j \in K \cup \{ h \}$, define a method q_j in each class as follows.

```
class $\sigma_i$
   method $q_j$
      new $\delta_\Sigma(\sigma_i, q_j)$; store $sym$;
      $R$; $setL$($sym$);
      $L$; $setR$($sym$);
      $sym$; $setL$($L$);
      $sym$; $setR$($R$);
      $sym$; $\delta_K(\sigma_i, q_j)$
```

For moving the head along the tape, we also have:

```
class $L$
   method $q_j$
      $L$; $\delta_K(\sigma_i, q_j)$
class $R$
   method $q_j$
      $R$; $\delta_K(\sigma_i, q_j)$
```

Every class has:

```
   method $setL$
      $arg$; store $L$
   method $setR$
      $arg$; store $R$
```

We have assumed that the initial graph contains as many vertices as the computation will require (that is, the simulated tape is long enough). A Turing machine, however, has an infinite tape. It is possible to enhance the methods given above so that they create new "tape" as necessary. The first and last vertex would be members of a different class, and moving beyond them would cause new vertices to be created.

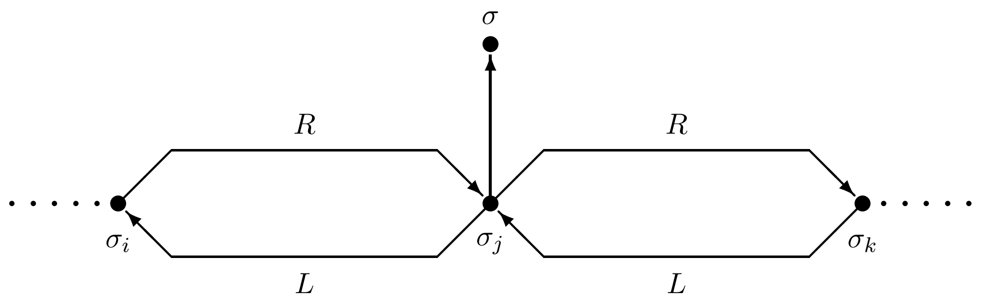

Figure 40: Simulating a Turing Machine

10.7 Inheritance

In this section, we outline a simple way of introducing inheritance into the model. It is necessary to change only the binding mechanism for method names.

We define a partial order, $\preceq$, on class names. If $\alpha \preceq \beta$, we say "class α inherits from class β". This definition is reflexive: every class inherits itself. We use $\prec$ to denote "proper" inheritance, between distinct classes. We introduce a function Ψ which generalizes Φ by returning a set of methods:

$$\Psi(\alpha, m) \quad = \quad \{\, \Phi(\gamma, m) \mid \gamma \in \Gamma \,\}$$

where Γ is the set of classes that satisfy these conditions:

$$\begin{array}{ll} \gamma \in \Gamma \text{ iff} & \Phi(\gamma, m) \text{ is defined, and} \\ & \alpha \preceq \gamma \text{ and} \\ & \forall \beta \in \mathcal{C} \,.\, \alpha \preceq \beta \prec \gamma \Rightarrow (\Phi(\beta, m) \text{ is not defined}) \end{array}$$

Each member of the set of programs returned by Ψ is a candidate for the body of the method to be evaluated. The set includes all methods provided by the ancestors of the current class that are not redefined. There are three possibilities.

1. If $|\Gamma| = 0$, there is no suitable method and evaluation fails.
2. If $|\Gamma| = 1$, there is exactly one applicable method, which is evaluated.
3. If $|\Gamma| > 1$, there are several applicable methods and the semantics must either forbid this situation or provide a strategy for choosing one of the methods.

11 Applying Semantics to Compiler Construction

A compiler translates a program written in a language $\mathcal{L}$ into the machine language of a particular processor. If we have formal definitions of $\mathcal{L}$ and the target language, it is possible in principle to generate the compiler automatically. Automatic generation of compilers has been a research topic for many years: the first "compiler-compilers" (which today we would call attribute grammar processors) were constructed in the fifties.

11.1 Compiler Structure

The process of compilation is traditionally split into several phases that we can describe using functions.

The source program is a text (or string) that belongs to the set $Char^*$. The first phase of the compiler (usually called the "scanner") scans this text, extracting tokens and ignoring white space and comments. Tokens include keywords, identifiers, operator symbols, punctuation, and so on. The output of this phase is a sequence of tokens: $Token^*$.

The tokens are then read by a parser that constructs an abstract syntax tree (AST) of type AST. The tree is called "abstract" because it models the structure of the program accurately but does not contain "noise" such as keywords and separators. Each node of the AST corresponds to a syntactic unit, such as a command or expression. The subtrees of a node are the component of the unit.

The next phase is semantic analysis, which both checks that the AST conforms to the static semantics of the source language and adds information to it. For example, the parse might convert the assignment statement

$$y := n + 1.0;$$

into the AST

$$assign\,(var\,(y), plus\,(var\,(n), float\_const\,(1.0))).$$

After semantic analysis, this tree would be expanded to

$$float\_assign(float\_var\,(y), float\_plus\,(int\_float\,(int\_var\,(n))), float\_const\,(1.0)\ ).$$

The result of semantic analysis is called a "decorated abstract syntax tree" and it belongs to the set AST^+.

The next phase, which may be omitted in a simple compiler, translates the decorated AST into a suitable intermediate representation, IR, of type IR.

It is a good idea to use an IR that is fairly independent of both the source language and the target language. Independence from the source language means that the same IR can be used in compilers for different languages; target language independence means that the IR can be used to generate code for different processors. Without a common IR, we would need $M \times N$ compilers for M source language and N processors; with a common IR, we need M "front ends" and N "back ends". To obtain a compiler, we combine a front-end and a back-end; the "ends" communicate using IR.

Finally, the code generator translates the intermediate representation into a sequence of instructions for the target processor, Instr^*. In summary:

$$\begin{aligned}\mathit{Scan} &: \mathit{Char}^* \to \mathit{Token}^* \\ \mathit{Parse} &: \mathit{Token}^* \to \mathit{AST} \\ \mathit{Sem} &: \mathit{AST} \to \mathit{AST}^+ \\ \mathit{Tran} &: \mathit{AST}^+ \to \mathit{IR} \\ \mathit{Gen} &: \mathit{IR} \to \mathit{Instr}^*\end{aligned}$$

Putting these together, we have a function $\mathit{Compile}\colon \mathit{Char}^* \to \mathit{Instr}^*$ defined by

$$\mathit{Compile} = \mathit{Gen} \circ \mathit{Tran} \circ \mathit{Sem} \circ \mathit{Parse} \circ \mathit{Scan}.$$

A practical compiler requires at least two additional features not discussed here: it must detect, report, and recover from errors; and it should perform optimizations at various places.

11.2 Compiler Generation

Some phases of the compiler can be generated by software tools. A scanner generator translates a description of the lexical structure of the language into a scanning procedure. A parser generator translates a description of the syntactic structure of the language into a parser.

The UNIX utility lex (GNU flex) is a familiar example of a scanner generator. The UNIX utility yacc (GNU bison) stands for "yet another compiler compiler" because it does more than generate a parser: its input is a grammar with attributes (written as chunks of C code) that are incorporated into the parser.

There are code generator generators (CGGs) for certain kinds of intermediate representation. Given a definition of the instruction set of the target machine, a CGG generates a program that translates an instance of the IR to optimized machine code. For example, the UNIX utility twig works by bottom-up tree-matching. The IR is a tree and the target architecture is represented by a number of tree patterns that represent the capabilities of the instruction set. The tree-matcher replaces parts of the IR by patterns until the entire tree represents a sequence of instructions.

The part of the compiler that has been least automated is the so-called "middle end": the semantic analyzer and the translator from the decorated abstract syntax tree to the intermediate representation.

11.3 Using Semantic Definitions

It should be possible to use the semantic definition of a language to generate the middle phases of a compiler. We can consider which kind of semantics would be most suitable for this task.

11.3.1 Operational Semantics

Operational semantics might seem to be the most natural choice. It is not hard to construct a compiler from operational semantics, but the results are likely to be not only inefficient but also hard to optimize.

Suppose that we used the operational semantics of L_1 (Section 2.3) as the basis for a compiler. The compiler would use the semantics to translate a program into a sequence of stack-machine instructions and would then generate code for each of these instructions. If the target machine had a simple stack-based architecture, the results might be acceptable. If, on the other hand, the target machine had a modern RISC architecture, with many registers, the simulated stack code would be very inefficient and hard to optimize.

11.3.2 Axiomatic Semantics

An axiomatic semantics enables us to reason about programs but is — intentionally — too abstract to use as the basis for a compiler.

11.3.3 Denotational Semantics

A denotational semantics contains all of the information needed to construct a compiler, and a number of researchers have built compiler generators using denotational semantics.

An early example was Mosses' (1975, 1979) Semantics Implementation System (SIS). The front-end is conventional. The back-end of the compiler is specified by semantic equations written in an untyped language called DSL. The result of putting a particular program through SIS is an expression in a language based on λ-calculus and called LAMB. The programs generated by SIS run extremely slowly in comparison to programs generated by conventional compilers.

Paulson's (1981, 1982) Semantics Processor (PSP) combines extended attribute grammars, denotational semantics, and a typed λ-calculus metalanguage. The grammar analyzer (GA) reads a semantic grammar and produces a language description file (LDF) containing parsing tables and attribute dependency information. The second component is a universal translator (UT), which becomes a compiler after translating a LDF. The output is code for an abstract stack machine (actually, Landin's (1964) SECD machine). PSP is quite efficient and has been used to generate practical compilers for subsets of Pascal and FORTRAN.

Wand's (1984) Semantic Prototyping System maps abstract syntax trees to λ-expressions encoded in Scheme. PSP is quite efficient and provides strong polymorphic type checking of the semantic specifications.

Figure 41 compares the performance of these systems, using PSP as a baseline. It is clear that SIS is inferior to PSP and SPS. Programs generated by PSP and SPS take roughly 1,000 times longer to execute than programs generated by conventional compilers.

Task	SIS	PSP	SPS
Compiler generation	29.9	1	
Program compilation	2.8	1	0.27
Program execution	165.0	1	0.90

Figure 41: Performance of Semantic Compiler Generators

Partial evaluation is an interesting technique suggested by denotational semantics. Suppose that we have a definition for a function $F(x, y)$ and that we know the value of one argument:

say, $x = k$. Then a *partial evaluation* of $F(x, y)$ yields a new function $F'(y) = F(k, y)$. Jones *et al.* (1985) implemented a partial evaluator called *mix* with the property that $mix\,(f, x) = f(x)$. Since *mix* uses partial evaluation, it produces an output even if the argument x is not present: that is, $mix\,(f) = \lambda x\,.\,f(x)$. If I is an interpreter for for a language L, then $mix\,(I)$ is a compiler for L. Furthermore, since $mix\,(mix, I) = mix\,(I)$, we can use $M = mix\,(mix)$ as a compiler generator and $mix\,(M)$ as a generator for compiler generators. All of these work but, needless to say, the generated code is spectacularly inefficient.

11.4 Efficient Compiler Generation

Peter Lee has analyzed the reasons for the inefficiency of compilers generated from denotational definitions (Lee 1989, Chapter 4). The main problem is *lack of separability*: the semantic equations are based on a model of computation and it is hard to separate the meaning of language features from the model.

Consider, for example, the program fragment

```
a := 1;
b := 2;
```

It is not hard to process these commands with a simple, direct semantics such as that of Section 2.6. But, as we have seen, a standard semantics for a practical language requires an environment, a store, and continuations. Expanding the program above with the appropriate semantic equations leads to the following expression (written in SML-like style):

```
( fn env .  fn cont .
   if (env *a*) = unbound
     then err
     else
       ( isloc ( fn (variable loc) .
         ( deref ( isRvalue (
           ( update loc (
             if (env *b*) = unbound
               then err
               else
                 ( isloc ( fn (variable loc) .
                   ( deref ( isRvalue (
                     ( update loc cont ) ) ) )
                   ( constant (integer 2)) ) )
                 (env *b*) ) ) ) ) )
         (constant (integer 1)) ) )
       (env *a*) )
```

The compiler generator must extract two simple integer assignments from this cumbersome expression.

There are some additional subtleties that are not apparent in this simple example. We know, for example, that it should not be necessary to make a copy of the entire store at each operation in case the copy will be needed later. But it is hard to infer this information from the semantic equations (it requires a tedious proof) and the compiler generator is unlikely to extract it. Consequently, it may generate code to save the values of variables that will never be used again.

A well-written denotational semantics is intended to describe a programming language using as few basic concepts as possible. A semantic description, for instance, might keep local variables, non-local variables, and formal parameters in a single environment. There is no way that a compiler generator can realize that this is a simplification, not a requirement. Consequently, the generated code will treat all of these variable classes in the same way, although a human compiler writer would probably deal with the in different ways to achieve efficient code.

One way to deal with these problems is to give a sequence of semantic definitions, successively introducing implementation details. This is essentially what Stoy (1977) is doing in his Chapters 10 through 13. The drawback of this approach is that we have to write many semantic definitions rather than one and, of course, we have to provide a (usually tedious) proof that each definition is equivalent to its predecessor in the sequence.

11.4.1 High Level Semantics

To overcome these problems, Mosses (1982) introduced Abstract Semantic Algebras (ASAs) and *action semantics*. (For an introduction to action semantics, see (Slonneger and Kurtz 1995, Chapter 13).) Mosses' ideas have been extended by Lee.

Lee's (1989, Chapter 5) *high level semantics* are similar to ASAs but are intended specifically for compiler generation. The idea is roughly as follows:

▷ Define an algebra $\mathcal{A}$ of semantic operations.

▷ Provide a high level semantics for the source language using the uninterpreted operations of $\mathcal{A}$.

▷ Provide a low level semantics by giving an interpretation of the operations of $\mathcal{A}$.

The only connection between the high level and low level semantics is the signatures of the operations of $\mathcal{A}$.

Suppose that the syntax of the source language L includes:

$$E \quad = \quad n \mid E_1 + E_2$$

In the algebra $\mathcal{A}$ we define the following operators

$$\begin{array}{rcl} \mathit{integer} & : & \texttt{Int} \to \mathit{Vaction} \\ \mathit{add} & : & \mathit{Vaction} \times \mathit{Vaction} \to \mathit{Vaction} \\ \mathcal{M} & : & \texttt{Exp} \to \mathit{Env} \to \mathit{Vaction} \end{array}$$

We do not define the type *Vaction*, nor do we give definitions of the operations *integer* and *add*. Thus the algebra is *abstract*. The intuitive idea is that a *Vaction* (or "value action") will be processed on the assumption that it is a value yielded by an expression.

Here are the semantic equations:

$$\begin{array}{rcl} \mathcal{M}\,[\![\, n \,]\!]\,\rho & = & \mathit{integer}(n) \\ \mathcal{M}\,[\![E_1 + E_2 \,]\!]\,\rho & = & \mathit{add}(\mathcal{M}\,[\![\, E_1 \,]\!]\rho,\ \mathcal{M}\,[\![\, E_2 \,]\!]\,\rho) \end{array}$$

At first glance, these equations look rather conventional. But there are important differences between thee equations and the standard equations.

- ▷ Expressions do not denote λ-terms but rather value-producing actions in the (so far undefined) domain *Vaction*.
- ▷ Model dependencies such as stores and continuations are not present. It is true that there is an environment but it, too, is uninterpreted, and at this level we know only its algebraic properties (for example, $\mathcal{M}$ requires an environment as its second argument).
- ▷ Fundamental operations of the source language (integer addition in the example) are represented by uninterpreted algebraic operations (*add* in the example). We are not committed to implementing addition using an equivalent of the mathematical operation $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$.
- ▷ The semantics is not *over-specified*. For example, we cannot infer from the equations that E_1 is evaluated before E_2, because this is a (hidden) property of *add*.
- ▷ The semantic equations describe only the *static* properties of programs (for example, the scope of variables, as determined by environments). There are no stores, continuations, or other manifestations of the dynamic semantics.

Lee calls this part of the semantics the *macrosemantics*. The result of applying the macrosemantics to a program is a *prefix operator term*, or POT.

The *microsemantics* provides an interpretation for the operators of the algebra $\mathcal{A}$. Here is a possible, but rather simple, microsemantics for the example above. We define the type

$$Vaction \quad : \quad State \to \texttt{Int}$$

and the operators

$$\begin{aligned} integer(n) &= \lambda\sigma \,.\, n \\ add(u, v) &= \lambda\sigma \,.\, (u\sigma + v\sigma). \end{aligned}$$

There can be more than one microsemantics for a given algebra. In this example, we have given the microsemantics as λ terms. We could also give it in the form of code for a simple abstract machine (such as a stack machine) or as code for an actual machine.

A practical macrosemantics typically includes checking that we did not perform above. Here is a more realistic macrosemantics for integer addition. The semantic function $\mathcal{M}$ has type $\texttt{Exp} \to Env \to (Type \times Vaction$.

```
M [[E 1 + E2]] ρ  =  let  $(t_1, v_1) = \mathcal{M}[\![E_1]\!]\rho$
                     and  $(t_2, v_2) = \mathcal{M}[\![E_2]\!]\rho$
                     in
                          if $t_1 = int\_type$ andalso $t_2 = int\_type$
                            then $(int\_type, add(v_1, v_2))$
                            else $\perp : (Type * Vaction)$
                     end
```

To obtain a semantics of variables, we must refine the concept of environment. In Lee's system, an environment is a mapping from identifiers to *modes*, where a mode describes the named entity.

$$\begin{aligned}
Env &= Id \to Mode \\
Mode &= \texttt{union}\,(none \mid const\colon Const \mid var\colon Var) \\
Const &= \texttt{Int} \\
Var &= Name \times Type \\
Type &= \texttt{union}\,(int\_type \mid bool\_type)
\end{aligned}$$

In the macrosemantics, however, we see only the types *Daction* (declaration actions) and *Taction* (type producing actions). The operators include

$$\begin{aligned}
DeclSimpVar &: Name \times Taction \to Daction \\
IntType &: Taction
\end{aligned}$$

The semantic function for declarations is

$$\mathcal{D} \;:\; \texttt{Dec} \to Env \to (Env \to Daction)$$

and the semantic equation for the declaration of an integer variable is

$$\begin{aligned}
\mathcal{D}[\![\texttt{int}\ \ id]\!]\rho \;=\; & \texttt{if} && NotDeclared(id, Env)\ \texttt{then} \\
& \texttt{let} && varname = name(id)\ \texttt{and} \\
& && mode = Var(varname, int\_type)\ \texttt{and} \\
& && newenv = addassoc(id, mode, ev) \\
& \texttt{in} && (newenv, decsimpvar(varname, int\_type)) \\
& \texttt{else} && error
\end{aligned}$$

Here is a small example; the source language is called ToyPL.

```
program example ("stdin", "stdout") is
   int a;
   begin
      a := 1;
      a := a + 2
   end;
```

This program is expanded by macrosemantic analysis into the POT shown in Figure 42. Although this expression is quite complicated (given the simplicity of the original program), it contains only *static* information about the program. To repeat, there are no environments, stores, or continuations. The next step depends on the microsemantics: the definitions of the various operators used in this expansion.

11.5 MESS

Lee's system is called MESS because high level definitions are **M**odular, **E**xtensible, **S**eparated **S**emantic specifications. Lee claims that the name has nothing to do with the state of the implementation. Figure 43 shows a block diagram of MESS.

```
Execute (
   Prog (
     "stdin",
     "stdout",
     Block (
       DeclSimpVar ( a, int_type ),
       StmtSeq (
         Assign ( Var (a, int_type ), integer(1) ),
         StmtSeq (
           Assign (
             Var (a, int_type ),
             Add (
               RValue (
                 Var (a, int_type ),
                 integer(2) ) ) )
           NullStmt ) ) ) ) )
```

Figure 42: The POT for a small program

The inputs to the system, shown on the left, consist of specifications of the front end (lexical and syntactic features of the source language), the macrosemantics, and the microsemantics. These inputs are processed by the three components of the MESS, which occupies the centre of the diagram.

FrEGe is a front-end generator written by Uwe Pleban (Lee's supervisor). Its output is the compiler front-end (scanner and parser) and a specification of the AST. The macrosemantic analyzer uses the macrosemantics specification, the AST specification, and a microsemantic interface provided by the microsemantics analyzer, to generate the compiler core — the heart of the compiler. Finally, the microsemantics analyzer uses the microsemantics specification to generate the compiler's code generator.

The compiler, shown at the right of the diagram, accepts source code, translates it into an AST, and passes the AST to the core. The core performs semantic processing and outputs POTs. The code generator translates the POTs to machine code.

Separability ensures that the compiler is retargetable. The microsemantics, as previously mentioned, can specify a result in the form of λ terms, abstract machine code, or code for an actual machine. In each case, the POTs used are the same.

MESS was implemented during the late 80s on a PC with an 8MHz 80286 processor and 640Kb of RAM. The programming languages used were Turbo-Pascal, Turbo-Prolog, and TI-Scheme.

The first compiler generated by MESS was a simple language called HypoPL. Compiler generation required about 5 minutes; compiling a small program (bubble sort) required about 4 seconds. Figure 44 compares the generated compiler to two commercial compilers on the same program. The DeSmet C compiler was one of the first implementations of C available on the PC.

Lee also used MESS to construct a compiler for SOL/C ("sort of like C"). Figure 45 shows the performance of this compiler in comparison with commercial compilers.

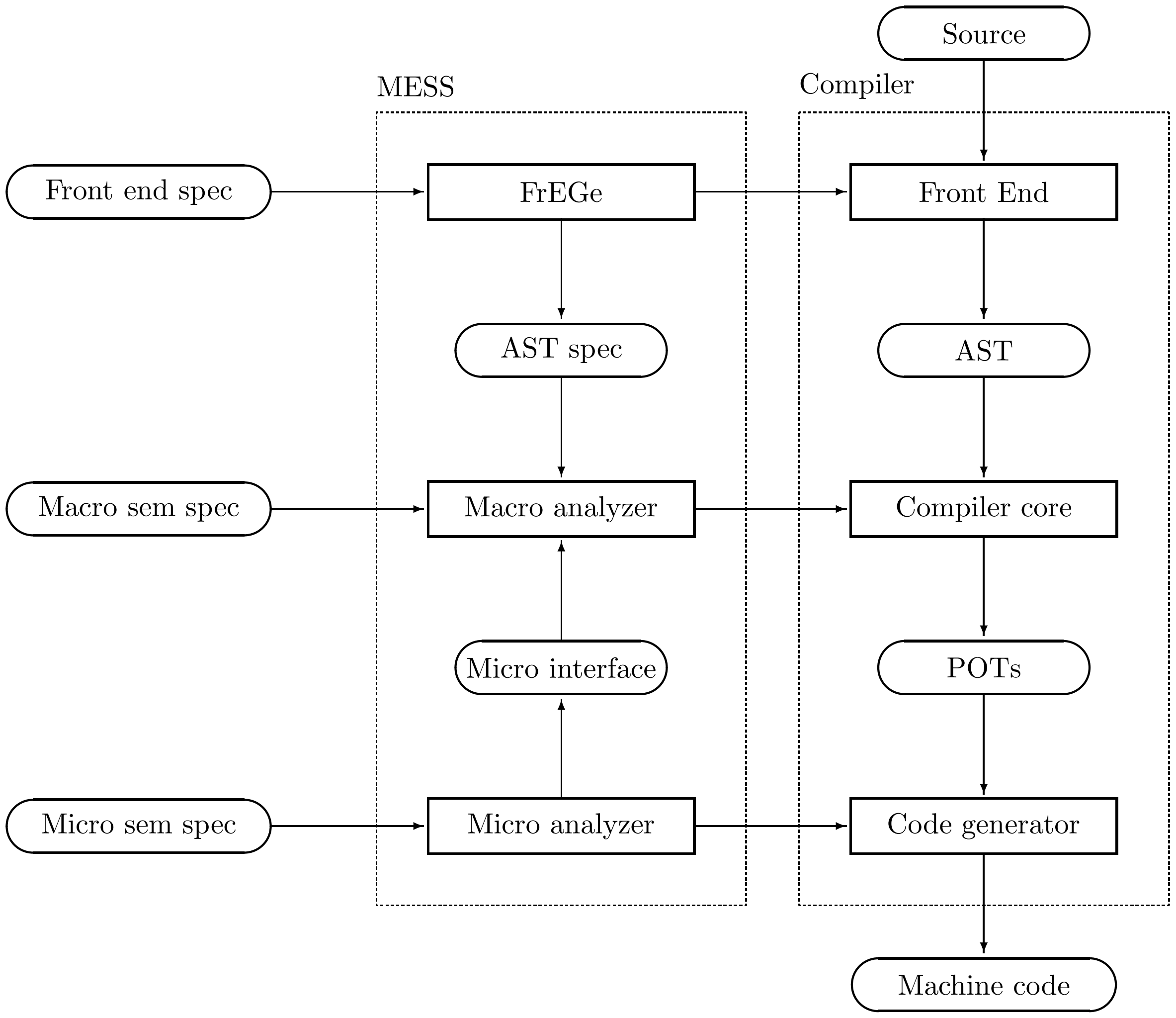

Figure 43: Peter Lee's MESS

Compiler	Size (bytes)	Time (secs)
MESS	230	29.7
DeSmet C	197	30.5
Turbo-Pascal	308	55.6

Figure 44: Performance of MESS

Program	Fib		Sort		Sieve		MatMult	
Compiler	Size	Time	Size	Time	Size	Time	Size	Time
MESS	153	34.1	227	15.3	198	8.6	339	19.8
DeSmet C	125	33.7	187	15.8	165	9.5	289	18.4
Turbo-Pascal	208	15.8	304	26.5	272	11.5	464	26.5

Figure 45: Performance of MESS

11.6 Conclusions

Lee suggests some extensions to MESS:

- ▷ it would be useful to be able to generate multi-pass (rather than only single-pass) compiler cores;
- ▷ it should be possible to provide the macrosemantics in modular form;
- ▷ the metalanguage should provide assignments and arrays (the current metalanguage is a pure functional language) to improve the efficiency of compiler generation.

Looking further ahead, Lee views MESS as the first step towards a *Language Designer's Workbench*, first proposed by Uwe Pleban, based on the following ideas.

- ▷ There should be a complete set of formal definition for a programming language, including syntax, static semantics, dynamic semantics, and a run-time model. These definitions provide the *standard of reference* for the language.
- ▷ It should be possible to test aspects of the language design by directly executing the formal definitions with minimal effort. In other words, it should be possible to gain some experience with a new language before writing a compiler for it.
- ▷ When the design is stable, it should be possible to generate an optimizing (or at least "non-pessimizing") compiler for it with little effort, again from the formal definitions.

Exercise 54 The following questions refer to the paper "Denotational Semantics for a Goal-Directed Language", by David A. Gudeman in *ACM Trans. Prog. Lang. and Sys.*, 14(1):107–125.

1. Describe the evaluation of this Icon program:

   s := `"abracadabra";`
   `write (find("a",`s`) < find("b",`s`))`

2. Write out the types *Cxt*, *Tran*, *Cont*, and *Store* in full.

3. Suppose that E is an Icon expression. Give the type of each entity in the definition

$$[\![E]\!]\, k \quad\equiv\quad k\, E.$$

4. Assume that `write` is a monogenic unary operator in the sense of Section 5. Give the type of each entity in the definition

$$[\![\texttt{write}\, E]\!] k \quad\equiv\quad [\![E\ ]\!] (\, [\![\texttt{write}\ ]\!] k\ ).$$

5. Assume that `write` E outputs the value of E and has no other effect. Give a suitable definition of $[\![$ `write` $]\!] k$.

6. Use the meaning that you have assigned to `write` to evaluate $[\![$ `write` $(3$ `to` $5)$ $]\!]\, k$.

7. Discuss the claims made in Section 10 and Section 11.

□

12 Bibliography

References

Ashcroft, E. and W. Wadge (1982). R_x for semantics. *ACM Trans. Programming Languages and Systems 4*(2), 283–294.

Dijkstra, E. W. (1976). *A Discipline of Programming.* Prentice Hall.

Floyd, R. (1967). Assigning meanings to programs. In *Proc. AMS Symp. Applied Mathematics*, Volume 19, Providence, R.I., pp. 19–31. American Mathematical Society.

Girard, J.-Y. (1989). *Proofs and Types.* Cambridge University Press. QA 9.54 G57X 1989.

Gunter, C. A. (1992). *Semantics of Programming Languages: Structures and Techniques.* MIT Press. QA 76.7 G86 1992.

Hoare, C. A. R. (1969, October). An axiomatic basis for computer programming. *Comm. ACM 12*(10), 576–580, 583.

Jones, N. D., P. Sestoft, and H. Søndergaard (1985). An experiment in partial evaluation: the generation of a compiler generator. In J. Jouannaud (Ed.), *Rewriting Techniques and Applications*, pp. 125–140. Published as *LNCS* 202, Springer-Verlag.

Landin, P. (1964). The mechanical evaluation of expressions. *J. ACM 6*, 308–20.

Landin, P. (1966, March). The next 700 programming languages. *Comm. ACM 9*(3), 157–64.

Lee, P. (1989). *Realistic Compiler Generation.* MIT Press.

MacQueen, D., G. Plotkin, and R. Sethi (1984, January). An ideal model for recursive polymorphic types. In *Proc. 11th ACM Symp. on Principles of Programming Languages*, pp. 165–174. ACM.

McCarthy, J. (1960, April). Recursive functions of symbolic expressions and their computation by machine, Part I. *Comm. ACM 3*(4), 184–195.

McCarthy, J. (1963). A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg (Eds.), *Computer Programming and Formal Systems*, pp. 33–70. Amsterdam: North-Holland.

McCarthy, J. (1981). History of LISP. In R. Wexelblat (Ed.), *History of Programming Languages*, pp. 173–196. Academic Press.

McCarthy, J., P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin (1962). *LISP 1.5 Programmer's Manual.* MIT Press.

Meyer, B. (1990). *Introduction to the Theory of Programming Languages.* Prentice Hall International. QA 76.7 M49 1988.

Milner, R. (1978). A theory of type polymorphism in programming. *J. Comp. and Sys. Science 17*, 348–375.

Mitchell, J. C. and G. Plotkin (1985). Abstract types have existential type. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pp. 36–51.

Mosses, P. D. (1975). *Mathematical semantics and compiler generation.* Ph. D. thesis, Oxford University.

Mosses, P. D. (1979). SIS — semantics implementation system. Technical Report DAIMI MD–30, Computer Science Deprtment, Aarhus University.

Mosses, P. D. (1982). Abstract semantic algebras! In D. Bjørner (Ed.), *Formal Descriptoin of Programming Concepts II*, pp. 63–68. IFIP IC–2.

Paulson, L. (1981). *A compiler generator for semantic grammars.* Ph. D. thesis, Department of Computer Science, Stanford University.

Paulson, L. (1982). A semantics-directed compiler generator. In *Proc. 9th ACM Symp. on Principles of Programming Languages*, pp. 224–239. ACM.

Penrose, R. (1989). *The Emperor's New Mind: Concerning Computers, Minds, and the Laws of Physics.* Oxford University Press.

Penrose, R. (1994). *Shadows of the Mind: a Search for the Missing Science of Consciousness.* Oxford University Press.

Robinson, J. (1965). A machine-oriented logic based on the resolution principle. *J. ACM 12*, 23–41.

Schmidt, D. A. (1986). *Denotational Semantics: A Methodology for Language Development.* W. C. Brown. QA 76.7 S34 1986.

Slonneger, K. and B. L. Kurtz (1995). *Formal Syntax and Semantics of Programming Languages.* Addison-Wesley.

Stoy, J. (1977). *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* MIT Press. QA 76.7 S74.

Tennent, R. D. (1977). Language design methods based on semantic principles. *Acta Informatica 8*(2), 97–112.

Tennent, R. D. (1981). *Principles of Programming Languages.* Prentice Hall International.

Wand, M. (1984). A semantic prototyping system. In *Proc. SIGPLAN 84 Symp. on Compiler Construction*, pp. 213–221. Published as *SIGPLAN Notices*, **19**(6).

Welsh, J., W. Sneeringer, and C. A. R. Hoare (1977). Ambiguities and insecurities in Pascal. *Software: Practice and Experience 7*, 685–696.

Winskel, G. (1993). *The Formal Semantics of Programming Languages.* MIT Press. QA 76.7 W555 1993.

Wirth, N. (1971, April). Program development by stepwise refinement. *Comm. ACM 14*(4), 221–227.