Chapter 5

Imperative Languages

Most sequential programming languages use a data structure that exists independently of any program in the language. The data structure isn't explicitly mentioned in the language's syntax, but it is possible to build phrases that access it and update it. This data structure is called the *store*, and languages that utilize stores are called *imperative.* The fundamental example of a store is a computer's primary memory, but file stores and data bases are also examples. The store and a computer program share an intimate relationship:

1. The store is critical to the evaluation of a phrase in a program. A phrase is understood in terms of how it handles the store, and the absence of a proper store makes the phrase nonexecutable.
2. The store serves as a means of communication between the different phrases in the program. Values computed by one phrase are deposited in the store so that another phrase may use them. The language's sequencing mechanism establishes the order of communication.
3. The store is an inherently ''large'' argument. Only one copy of store exists at any point during the evaluation.

In this chapter, we study the store concept by examining three imperative languages. You may wish to study any subset of the three languages. The final section of the chapter presents some variants on the store and how it can be used.

5.1 A LANGUAGE WITH ASSIGNMENT

The first example language is a declaration-free Pascal subset. A program in the language is a sequence of *commands.* Stores belong to the domain *Store* and serve as arguments to the valuation function:

$$\mathbf{C}\text{: Command} \to \mathit{Store}_{\perp} \to \mathit{Store}_{\perp}$$

The purpose of a command is to produce a new store from its store argument. However, a command might not terminate its actions upon the store— it can ''loop.'' The looping of a command $[\![C]\!]$ with store s has semantics $\mathbf{C}[\![C]\!]s = \perp$. (This explains why the *Store* domain is lifted: $\perp$ is a possible answer.) The primary property of nontermination is that it creates a nonrecoverable situation. Any commands $[\![C']\!]$ following $[\![C]\!]$ in the evaluation sequence will not evaluate. This suggests that the function $\mathbf{C}[\![C']\!]: \mathit{Store}_{\perp} \to \mathit{Store}_{\perp}$ be strict; that is, given a nonrecoverable situation, $\mathbf{C}[\![C']\!]$ can do nothing at all. Thus, command composition is $\mathbf{C}[\![C_1;C_2]\!] = \mathbf{C}[\![C_2]\!] \circ \mathbf{C}[\![C_1]\!]$.

Figure 5.1 presents the semantic algebras for the imperative language. The *Store* domain models a computer store as a mapping from the identifiers of the language to their values. The

Figure 5.1

I. Truth Values
Domain $t \in Tr = \mathbb{B}$
Operations

$true, false : Tr$

$not : Tr \to Tr$

II. Identifiers
Domain $i \in Id =$ Identifier

III. Natural Numbers
Domain $n \in Nat = \mathbb{N}$
Operations

$zero, one, \cdots : Nat$

$plus : Nat \times Nat \to Nat$

$equals : Nat \times Nat \to Tr$

IV. Store
Domain $s \in Store = Id \to Nat$
Operations

$newstore : Store$
$newstore = \lambda i.\ zero$

$access : Id \to Store \to Nat$
$access = \lambda i.\ \lambda s.\ s(i)$

$update : Id \to Nat \to Store \to Store$
$update = \lambda i.\lambda n.\lambda s.\ [\, i \mapsto n \,]s$

operations upon the store include a constant for creating a new store, an operation for accessing a store, and an operation for placing a new value into a store. These operations are exactly those described in Example 3.11 of Chapter 3.

The language's definition appears in Figure 5.2.

The valuation function **P** states that the meaning of a program is a map from an input number to an answer number. Since nontermination is possible, $\perp$ is also a possible ''answer,'' hence the rightmost codomain of **P** is $Nat_\perp$ rather than just *Nat*. The equation for **P** says that the input number is associated with identifier $[\![A]\!]$ in a new store. Then the program body is evaluated, and the answer is extracted from the store at $[\![Z]\!]$.

The clauses of the **C** function are all strict in their use of the store. Command composition works as described earlier. The conditional commands are choice functions. Since the

Figure 5.2

Abstract syntax:

P∈ Program
C∈ Command
E∈ Expression
B∈ Boolean-expr
I∈ Identifier
N∈ Numeral

P ::= C.

C ::= C_1;C_2 | **if** B **then** C | **if** B **then** C_1 **else** C_2 | I:=E | **diverge**

E ::= E_1+E_2 | I | N

B ::= E_1=E_2 | ¬B

Semantic algebras:
(defined in Figure 5.1)

Valuation functions:

P: Program→ *Nat*→ $Nat_\perp$
P⟦C.⟧ = λ*n*. let *s*= (*update*⟦A⟧ *n newstore*) in
let *s*′ = **C**⟦C⟧*s* in (*access*⟦Z⟧ *s*′)

C: Command→ $Store_\perp$→ $Store_\perp$
C⟦C_1;C_2⟧ = $\underline{\lambda}$*s*. **C**⟦C_2⟧ (**C**⟦C_1⟧*s*)
C⟦**if** B **then** C⟧ = $\underline{\lambda}$*s*. **B**⟦B⟧*s*→ **C**⟦C⟧*s*[] *s*
C⟦**if** B **then** C_1 **else** C_2⟧ = $\underline{\lambda}$*s*. **B**⟦B⟧*s*→ **C**⟦C_1⟧*s*[] **C**⟦C_2⟧*s*
C⟦I:=E⟧ = $\underline{\lambda}$*s*. *update*⟦I⟧ (**E**⟦E⟧*s*) *s*
C⟦**diverge**⟧ = $\underline{\lambda}$*s*. ⊥

E: Expression→ *Store*→ *Nat*
E⟦E_1+E_2⟧ = λ*s*. **E**⟦E_1⟧*s plus* **E**⟦E_2⟧*s*
E⟦I⟧ = λ*s*. *access* ⟦I⟧ *s*
E⟦N⟧ = λ*s*. **N**⟦N⟧

B: Boolean-expr→ *Store*→ *Tr*
B⟦E_1=E_2⟧ = λ*s*. **E**⟦E_1⟧*s equals* **E**⟦E_2⟧*s*
B⟦¬B⟧ = λ*s*. *not*(**B**⟦B⟧*s*)

N: Numeral→ *Nat* (*omitted*)

expression ($e_1 \rightarrow e_2$ [] e_3) is nonstrict in arguments e_2 and e_3, the value of **C**⟦**if** B **then** C⟧s is s when **B**⟦B⟧s is *false*, even if **C**⟦C⟧$s = \underline{\perp}$. The assignment statement performs the expected update; the ⟦**diverge**⟧ command causes nontermination.

The **E** function also needs a store argument, but the store is used in a ''read only'' mode. **E**'s functionality shows that an expression produces a number, not a new version of store; the store is not updated by an expression. The equation for addition is stated so that the order of evaluation of ⟦E_1⟧ and ⟦E_2⟧ is not important to the final answer. Indeed, the two expressions might even be evaluated in parallel. A strictness check of the store is not needed, because **C** has already verified that the store is proper prior to passing it to **E**.

Here is the denotation of a sample program with the input *two*:

P⟦Z:=1; **if** A=0 **then diverge**; Z:=3.⟧(*two*)
= let s = (*update*⟦A⟧ *two newstore*) in
  let s' = **C**⟦Z:=1; **if** A=0 **then diverge**; Z:=3⟧s
  in *access*⟦Z⟧ s'

Since (*update*⟦A⟧ *two newstore*) is ([⟦A⟧↦ *two*] *newstore*), that is, the store that maps ⟦A⟧ to *two* and all other identifiers to *zero*, the above expression simplifies to:

let s' = **C**⟦Z:=1; **if** A=0 **then diverge**; Z:=3⟧ ([⟦A⟧↦ *two*] *newstore*)
in *access*⟦Z⟧ s'

From here on, we use s_1 to stand for ([⟦A⟧→ *two*] *newstore*). Working on the value bound to s' leads us to derive:

C⟦Z:=1; **if** A=0 **then diverge**; Z:=3⟧s_1
= ($\underline{\lambda}s$. **C**⟦**if** A=0 **then diverge**; Z:=3⟧ (**C**⟦Z:=1⟧s))s_1

The store s_1 is a proper value, so it can be bound to s, giving:

C⟦**if** A=0 **then diverge**; Z:=3⟧ (**C**⟦Z:=1⟧s_1)

We next work on **C**⟦Z:=1⟧s_1:

C⟦Z:=1⟧s_1
= ($\underline{\lambda}s$. *update*⟦Z⟧ (**E**⟦1⟧s) s) s_1
= *update*⟦Z⟧ (**E**⟦1⟧s_1) s_1
= *update*⟦Z⟧ (**N**⟦1⟧) s_1
= *update*⟦Z⟧ *one* s_1
= [⟦Z⟧↦ *one*] [⟦A⟧↦ *two*] *newstore*

which we call s_2. Now:

C⟦**if** A=0 **then diverge**; Z:=3⟧s_2
= ($\underline{\lambda}s$. **C**⟦Z:=3⟧ (($\underline{\lambda}s$. **B**⟦A=0⟧$s \rightarrow$ **C**⟦**diverge**⟧s [] s)s))s_2
= **C**⟦Z:=3⟧ (($\underline{\lambda}s$. **B**⟦A=0⟧$s \rightarrow$ **C**⟦**diverge**⟧s [] s)s_2)
= **C**⟦Z:=3⟧ (**B**⟦A=0⟧$s_2 \rightarrow$ **C**⟦**diverge**⟧s_2 [] s_2)

Note that **C**⟦**diverge**⟧s_2 = ($\underline{\lambda}s$. $\underline{\perp}$)s_2 = $\underline{\perp}$, so nontermination is the result if the test has value

true. Simplifying the test, we obtain:

$$\begin{aligned}&\mathbf{B}[\![\text{A=0}]\!]s_2 = (\lambda s.\ \mathbf{E}[\![\text{A}]\!]s\ \mathit{equals}\ \mathbf{E}[\![0]\!]s)s_2\\ &= \mathbf{E}[\![\text{A}]\!]s_2\ \mathit{equals}\ \mathbf{E}[\![0]\!]s_2\\ &= (\mathit{access}[\![\text{A}]\!]\ s_2)\ \mathit{equals}\ \mathit{zero}\end{aligned}$$

Examining the left operand, we see that:

$$\begin{aligned}&\mathit{access}[\![\text{A}]\!]\ s_2\\ &= s_2[\![\text{A}]\!]\\ &= (\,[\,[\![\text{Z}]\!] \mapsto \mathit{one}\,]\,[\,[\![\text{A}]\!] \mapsto \mathit{two}\,]\,\mathit{newstore})\,[\![\text{A}]\!]\\ &= (\,[\,[\![\text{A}]\!] \mapsto \mathit{two}\,]\,\mathit{newstore})\,[\![\text{A}]\!] \qquad \text{(why?)}\\ &= \mathit{two}\end{aligned}$$

Thus, $\mathbf{B}[\![\text{A=0}]\!]s_2 = \mathit{false}$, implying that $\mathbf{C}[\![\textbf{if A=0 then diverge}]\!]s_2 = s_2$. Now:

$$\begin{aligned}&\mathbf{C}[\![\text{Z:=3}]\!]s_2\\ &= [\,[\![\text{Z}]\!] \mapsto \mathit{three}\,]s_2\end{aligned}$$

The denotation of the entire program is:

$$\begin{aligned}&\text{let } s' = [\,[\![\text{Z}]\!] \mapsto \mathit{three}\,]s_2 \text{ in } \mathit{access}[\![\text{Z}]\!]\ s'\\ &= \mathit{access}[\![\text{Z}]\!]\ [\,[\![\text{Z}]\!] \mapsto \mathit{three}\,]s_2\\ &= (\,[\,[\![\text{Z}]\!] \mapsto \mathit{three}\,]s_2)\,[\![\text{Z}]\!]\\ &= \mathit{three}\end{aligned}$$

We obtain a much different denotation when the input number is *zero*:

$$\begin{aligned}&\mathbf{P}[\![\text{Z:=1; }\textbf{if A=0 then diverge}\text{; Z:=3.}]\!](\mathit{zero})\\ &= \text{let } s' = \mathbf{C}[\![\text{Z:=1; }\textbf{if A=0 then diverge}\text{; Z:=3}]\!]s_3 \text{ in } \mathit{access}[\![\text{Z}]\!]\ s'\end{aligned}$$

where $s_3 = [\,[\![\text{A}]\!] \mapsto \mathit{zero}\,]\,\mathit{newstore}$. Simplifying the value bound to s' leads to:

$$\begin{aligned}&\mathbf{C}[\![\text{Z:=1; }\textbf{if A=0 then diverge}\text{; Z:=3}]\!]s_3\\ &= \mathbf{C}[\![\textbf{if A=0 then diverge}\text{; Z:=3}]\!]s_4\end{aligned}$$

where $s_4 = [\,[\![\text{Z}]\!] \mapsto \mathit{one}\,]s_3$. As for the conditional, we see that:

$$\begin{aligned}&\mathbf{B}[\![\text{A=0}]\!]s_4 \rightarrow \mathbf{C}[\![\textbf{diverge}]\!]s_4\ [\,]\ s_4\\ &= \mathit{true} \rightarrow \mathbf{C}[\![\textbf{diverge}]\!]s_4\ [\,]\ s_4\\ &= \mathbf{C}[\![\textbf{diverge}]\!]s_4\\ &= (\underline{\lambda} s.\ \bot)s_4\\ &= \bot\end{aligned}$$

So the value bound to s' is $\mathbf{C}[\![\text{Z:=3}]\!]\bot$. But $\mathbf{C}[\![\text{Z:=3}]\!]\bot = (\underline{\lambda} s.\ \mathit{update}[\![\text{Z}]\!]\ (\mathbf{E}[\![3]\!]s)\ s)\bot = \bot$. Because of the strict abstraction, the assignment isn't performed. The denotation of the program is:

$$\text{let } s' = \bot \text{ in } \mathit{access}[\![\text{Z}]\!]\ s'$$

which simplifies directly to $\bot$. (Recall that the form (let $x = e_1$ in e_2) represents $(\underline{\lambda} x.\, e_2)e_1$.) The undefined store forces the value of the entire program to be undefined.

The denotational definition is also valuable for proving properties such as program equivalence. As a simple example, we show for distinct identifiers ⟦X⟧ and ⟦Y⟧ that the command **C**⟦X:=0; Y:=X+1⟧ has the same denotation as **C**⟦Y:=1; X:=0⟧. The proof strategy goes as follows: since both commands are functions in the domain $Store_\bot \to Store_\bot$, it suffices to prove that the two functions are equal by showing that both produce same answers from same arguments. (This is because of the principle of extensionality mentioned in Section 3.2.3.) First, it is easy to see that if the store argument is $\bot$, both commands produce the answer $\bot$. If the argument is a proper value, let us call it *s* and simplify:

C⟦X:=0; Y:=X+1⟧*s*
= **C**⟦Y:=X+1⟧ (**C**⟦X:=0⟧*s*)
= **C**⟦Y:=X+1⟧ ([⟦X⟧↦ *zero*]*s*)
= *update*⟦Y⟧ (**E**⟦X+1⟧ ([⟦X⟧↦ *zero*]*s*)) ([⟦X⟧↦ *zero*]*s*)
= *update*⟦Y⟧ *one* [⟦X⟧↦ *zero*]*s*
= [⟦Y⟧↦ *one*] [⟦X⟧↦ *zero*]*s*

Call this result s_1. Next:

C⟦Y:=1; X:=0⟧*s*
= **C**⟦X:=0⟧ (**C**⟦Y:=1⟧*s*)
= **C**⟦X:=0⟧ ([⟦Y⟧↦ *one*]*s*)
= [⟦X⟧↦ *zero*] [⟦Y⟧↦ *one*]*s*

Call this result s_2. The two values are defined stores. Are they the *same* store? It is not possible to simplify s_1 into s_2 with the simplification rules. But, recall that stores are themselves functions from the domain $Id \to Nat$. To prove that the two stores are the same, we must show that each produces the same number answer from the same identifier argument. There are three cases to consider:

1. The argument is ⟦X⟧: then s_1⟦X⟧ = ([⟦Y⟧↦ *one*] [⟦X⟧↦ *zero*]*s*) ⟦X⟧ = ([⟦X⟧↦ *zero*]*s*) ⟦X⟧ = *zero*; and s_2 ⟦X⟧ = ([⟦X⟧↦ *zero*] [⟦Y⟧↦ *one*]*s*) ⟦X⟧ = *zero*.
2. The argument is ⟦Y⟧: then s_1⟦Y⟧ = ([⟦Y⟧↦ *one*] [⟦X⟧↦ *zero*]*s*) ⟦Y⟧ = *one*; and s_2 ⟦Y⟧ = ([⟦X⟧↦ *zero*] [⟦Y⟧↦ *one*]*s*) ⟦Y⟧ = ([⟦Y⟧↦ *one*]*s*) ⟦Y⟧ = *one*.
3. The argument is some identifier ⟦I⟧ other than ⟦X⟧ or ⟦Y⟧: then s_1⟦I⟧ = *s*⟦I⟧ and s_2⟦I⟧ = *s*⟦I⟧.

Since s_1 and s_2 behave the same for all arguments, they are the same function. This implies that **C**⟦X:=0; Y:=X+1⟧ and **C**⟦Y:=1; X:=0⟧ are the same function, so the two commands are equivalent. Many proofs of program properties require this style of reasoning.

5.1.1 Programs Are Functions

The two sample simplification sequences in the previous section were operational-like: a program and its input were computed to an answer. This makes the denotational definition behave like an operational semantics, and it is easy to forget that functions and domains are even involved. Nonetheless, it is possible to study the denotation of a program *without* supplying sample input, a feature that is not available to operational semantics. This broader view emphasizes that the denotation of a program is a *function.*

Consider again the example ⟦Z:=1; **if** A=0 **then diverge**; Z:=3⟧. What is its meaning? It's a function from *Nat* to $Nat_\perp$:

$$
\begin{aligned}
&\mathbf{P}[\![\text{Z:=1; }\textbf{if}\text{ A=0 }\textbf{then diverge}\text{; Z:=3.}]\!]\\
&= \lambda n.\ \text{let } s = \mathit{update}[\![\text{A}]\!]\ n\ \mathit{newstore} \text{ in}\\
&\quad \text{let } s' = \mathbf{C}[\![\text{Z:=1; }\textbf{if}\text{ A=0 }\textbf{then diverge}\text{; Z:=3}]\!]s\\
&\quad \text{in } \mathit{access}[\![\text{Z}]\!]\ s'\\
&= \lambda n.\ \text{let } s = \mathit{update}[\![\text{A}]\!]\ n\ \mathit{newstore} \text{ in}\\
&\quad \text{let } s' = (\underline{\lambda} s.\ (\underline{\lambda} s.\ \mathbf{C}[\![\text{Z:=3}]\!]\ (\mathbf{C}[\![\textbf{if}\text{ A=0 }\textbf{then diverge}]\!]s))s)(\mathbf{C}[\![\text{Z:=1}]\!]s)\\
&\quad \text{in } \mathit{access}[\![\text{Z}]\!]\ s'\\
&= \lambda n.\ \text{let } s = \mathit{update}[\![\text{A}]\!]\ n\ \mathit{newstore} \text{ in}\\
&\quad \text{let } s' = (\underline{\lambda} s.\ (\underline{\lambda} s.\ \mathit{update}[\![\text{Z}]\!]\ \mathit{three}\ s)\\
&\qquad ((\underline{\lambda} s.\ (\mathit{access}[\![\text{A}]\!]\ s)\ \mathit{equals}\ \mathit{zero} \rightarrow (\underline{\lambda} s.\ \perp)s\ [\!]\ s)s))\\
&\qquad\quad ((\underline{\lambda} s.\ \mathit{update}[\![\text{Z}]\!]\ \mathit{one}\ s)s)\\
&\quad \text{in } \mathit{access}[\![\text{Z}]\!]\ s'
\end{aligned}
$$

which can be restated as:

$$
\begin{aligned}
&\lambda n.\ \text{let } s = \mathit{update}[\![\text{A}]\!]\ n\ \mathit{newstore} \text{ in}\\
&\quad \text{let } s' = (\text{let } s'_1 = \mathit{update}[\![\text{Z}]\!]\ \mathit{one}\ s \text{ in}\\
&\qquad\quad \text{let } s'_2 = (\mathit{access}[\![\text{A}]\!]\ s'_1)\ \mathit{equals}\ \mathit{zero} \rightarrow (\underline{\lambda} s.\ \perp)s'_1\ [\!]\ s'_1\\
&\qquad\quad \text{in } \mathit{update}[\![\text{Z}]\!]\ \mathit{three}\ s'_2)\\
&\quad \text{in } \mathit{access}[\![\text{Z}]\!]\ s'
\end{aligned}
$$

The simplifications taken so far have systematically replaced syntax constructs by their function denotations; all syntax pieces are removed (less the identifiers). The resulting expression denotes the meaning of the program. (A comment: it is proper to be concerned why a phrase such as **E**⟦0⟧*s* was simplified to *zero* even though the value of the store argument *s* is unknown. The simplification works because *s* is an argument bound to $\underline{\lambda} s$. Any undefined stores are ''trapped'' by $\underline{\lambda} s$. Thus, within the scope of the $\underline{\lambda} s$, all occurrences of *s* represent defined values.)

The systematic mapping of syntax to function expressions resembles compiling. The function expression certainly does resemble compiled code, with its occurrences of tests, accesses, and updates. But it is still a function, mapping an input number to an output number.

As it stands, the expression does not appear very attractive, and the intuitive meaning of the original program does not stand out. The simplifications shall proceed further. Let s_0 be (*update*⟦A⟧ *n newstore*). We simplify to:

$$\lambda n.\ \text{let } s' = (\text{let } s'_1 = \mathit{update}[\![\mathrm{Z}]\!]\ \mathit{one}\ s_0 \text{ in}$$
$$\text{let } s'_2 = (\mathit{access}[\![\mathrm{A}]\!]\ s'_1)\ \mathit{equals}\ \mathit{zero} \rightarrow (\underline{\lambda} s.\ \bot) s'_1\ []\ s'_1$$
$$\text{in } \mathit{update}[\![\mathrm{Z}]\!]\ \mathit{three}\ s'_2)$$
$$\text{in } \mathit{access}[\![\mathrm{Z}]\!]\ s'$$

We use s_1 for $(\mathit{update}[\![\mathrm{Z}]\!]\ \mathit{one}\ s_0)$; the conditional in the value bound to s'_2 is:

$$(\mathit{access}[\![\mathrm{A}]\!]\ s_1)\ \mathit{equals}\ \mathit{zero} \rightarrow \bot\ []\ s_1$$
$$= n\ \mathit{equals}\ \mathit{zero} \rightarrow \bot\ []\ s_1$$

The conditional can be simplified no further. We can make use of the following property; ''for $e_2 \in \mathit{Store}_\bot$ such that $e_2 \neq \bot$, let $s = (e_1 \rightarrow \bot\ []\ e_2)$ in e_3 equals $e_1 \rightarrow \bot\ []\ [e_2/s]e_3$.'' (The proof is left as an exercise.) It allows us to state that:

$$\text{let } s'_2 = (n\ \mathit{equals}\ \mathit{zero} \rightarrow \bot\ []\ s_1) \text{ in } \mathit{update}[\![\mathrm{Z}]\!]\ \mathit{three}\ s'_2$$
$$= n\ \mathit{equals}\ \mathit{zero} \rightarrow \bot\ []\ \mathit{update}[\![\mathrm{Z}]\!]\ \mathit{three}\ s_1$$

This reduces the program's denotation to:

$$\lambda n.\ \text{let } s' = (n\ \mathit{equals}\ \mathit{zero} \rightarrow \bot\ []\ \mathit{update}[\![\mathrm{Z}]\!]\ \mathit{three}\ s_1) \text{ in } \mathit{access}[\![\mathrm{Z}]\!]\ s'$$

The property used above can be applied a second time to show that this expression is just:

$$\lambda n.\ n\ \mathit{equals}\ \mathit{zero} \rightarrow \bot\ []\ \mathit{access}[\![\mathrm{Z}]\!]\ (\mathit{update}[\![\mathrm{Z}]\!]\ \mathit{three}\ s_1)$$

which is:

$$\lambda n.\ n\ \mathit{equals}\ \mathit{zero} \rightarrow \bot\ []\ \mathit{three}$$

which is the intuitive meaning of the program!

This example points out the beauty in the denotational semantics method. It extracts the *essence* of a program. What is startling about the example is that the primary semantic argument, the store, disappears completely, because it does not figure in the input-output relation that the program describes. This program does indeed denote a function from *Nat* to $\mathit{Nat}_\bot$.

Just as the replacement of syntax by function expressions resembles compilation, the internal simplification resembles compile-time code optimization. When more realistic languages are studied, such ''optimizations'' will be useful for understanding the nature of semantic arguments.

5.2 AN INTERACTIVE FILE EDITOR

The second example language is an interactive file editor. We define a *file* to be a list of records, where the domain of records is taken as primitive. The file editor makes use of two levels of store: the primary store is a component holding the file edited upon by the user, and the secondary store is a system of text files indexed by their names. The domains are listed in Figure 5.3.

The edited files are values from the *Openfile* domain. An opened file $r_1, r_2, \cdots, r_{last}$ is

Figure 5.3

IV. Text file Domain $f \in File = Record^*$

V. File system
  Domain $s \in File\text{-}system = Id \to File$
  Operations
    $access: Id \times File\text{-}system \to File$
      $access = \lambda(i,s).\ s(i)$
    $update: Id \times File \times File\text{-}system \to File\text{-}system$
      $update = \lambda(i,f,s).\ [\,i \mapsto f\,]s$

VI. Open file
  Domain $p \in Openfile = Record^* \times Record^*$
  Operations
    $newfile: Openfile$
      $newfile = (nil, nil)$
    $copyin: File \to Openfile$
      $copyin = \lambda f.\ (nil, f)$
    $copyout: Openfile \to File$
      $copyout = \lambda p.$ ''appends $fst(p)$ to $snd(p)$— defined later''
    $forwards: Openfile \to Openfile$
      $forwards = \lambda(front, back).\ null\ back \to (front, back)$
        [] $((hd\ back)\ cons\ front,\ (tl\ back))$
    $backwards: Openfile \to Openfile$
      $backwards = \lambda(front, back).\ null\ front \to (front, back)$
        [] $(tl\ front,\ (hd\ front)\ cons\ back)$
    $insert: Record \times Openfile \to Openfile$
      $insert = \lambda(r,\ (front, back)).\ null\ back \to (front,\ r\ cons\ back)$
        [] $((hd\ back)\ cons\ front),\ r\ cons\ (tl\ back))$
    $delete: Openfile \to Openfile$
      $delete = \lambda(front, back).\ (front,\ (null\ back \to back\ []\ tl\ back))$
    $at\text{-}first\text{-}record: Openfile \to Tr$
      $at\text{-}first\text{-}record = \lambda(front, back).\ null\ front$
    $at\text{-}last\text{-}record: Openfile \to Tr$
      $at\text{-}last\text{-}record = \lambda(front, back).\ null\ back \to true$
        [] $(null\ (tl\ back) \to true\ []\ false)$
    $isempty: Openfile \to Tr$
      $isempty = \lambda(front, back).\ (null\ front)\ and\ (null\ back)$

represented by two lists of text records; the lists break the file open in the middle:

$$r_{i-1} \cdots r_2\ r_1 \qquad r_i\ r_{i+1} \cdots r_{last}$$

r_i is the ''current'' record of the opened file. Of course, this is not the only representation of an opened file, so it is important that all operations that depend on this representation be grouped with the domain definition. There are a good number of them. *Newfile* represents a file with no records. *Copyin* takes a file from the file system and organizes it as:

$$r_1\ r_2 \cdots r_{last}$$

Record r_1 is the current record of the file. Operation *copyout* appends the two lists back together. A definition of the operation appears in the next chapter.

The *forwards* operation makes the record following the current record the new current record. Pictorially, for:

$$r_{i-1} \cdots r_2\ r_1 \qquad r_i\ r_{i+1}\ r_{last}$$

a forwards move produces:

$$r_i\ r_{i-1} \cdots r_2\ r_1 \qquad r_{i+1} \cdots r_{last}$$

Backwards performs the reverse operation. *Insert* places a record r behind the current record; an insertion of record r' produces:

$$r_i \cdots r_2\ r_1 \qquad r'\ r_{i+1} \cdots r_{last}$$

The newly inserted record becomes current. *Delete* removes the current record. The final three operations test whether the first record in the file is current, the last record in the file is current, or if the file is empty.

Figure 5.4 gives the semantics of the text editor.

Since all of the file manipulations are done by the operations for the *Openfile* domain, the semantic equations are mainly concerned with trapping unreasonable user requests. They also model the editor's output log, which echoes the input commands and reports errors.

The **C** function produces a line of terminal output and a new open file from its open file argument. For user commands such as [[**newfile**]], the action is quite simple. Others, such as [[**moveforward**]], can generate error messages, which are appended to the output log. For example:

C[[**delete**]](*newfile*)
= let $(k', p') = isempty\,(newfile) \rightarrow$ (''error: file is empty'', *newfile*)

Figure 5.4

Abstract syntax:

P∈ Program-session
S∈ Command-sequence
C∈ Command
R∈ Record
I∈ Identifier

P ::= **edit** I **cr** S

S ::= C **cr** S | **quit**

C ::= **newfile** | **moveforward** | **moveback** | **insert** R | **delete**

Semantic algebras:

I. Truth values
Domain $t \in Tr$
Operations

$true, false : Tr$

$and : Tr \times Tr \rightarrow Tr$

II. Identifiers
Domain $i \in Id =$ Identifier

III. Text records
Domain $r \in Record$

IV. - VI. defined in Figure 5.3

VII. Character Strings (defined in Example 3.3 of Chapter 3)

VIII. Output terminal log
Domain $l \in Log = String^*$

Valuation functions:

P: Program-session $\rightarrow$ *File-system* $\rightarrow$ (*Log* $\times$ *File-system*)

$\mathbf{P}[\![\textbf{edit I cr S}]\!] = \lambda s.$ let $p = copyin(access([\![\mathrm{I}]\!], s))$ in
("edit I" *cons* $fst(\mathbf{S}[\![\mathrm{S}]\!]p), update([\![\mathrm{I}]\!], copyout(snd(\mathbf{S}[\![\mathrm{S}]\!]p)), s)$)

S: Command-sequence $\rightarrow$ *Openfile* $\rightarrow$ (*Log* $\times$ *Openfile*)

$\mathbf{S}[\![\mathrm{C}\ \textbf{cr}\ \mathrm{S}]\!] = \lambda p.$ let $(l', p') = \mathbf{C}[\![\mathrm{C}]\!]p$ in $((l'\ cons\ fst(\mathbf{S}[\![\mathrm{S}]\!]p')), snd(\mathbf{S}[\![\mathrm{S}]\!]p'))$

$\mathbf{S}[\![\textbf{quit}]\!] = \lambda p.$ ("quit" *cons nil*, *p*)

Figure 5.4 (continued)

```
C: Command → Openfile → (String × Openfile)
  C[[newfile]] = λp. ("newfile", newfile)
  C[[moveforward]] = λp. let (k', p') = isempty(p) → ("error: file is empty", p)
                                 [] ( at-last-record(p) → ("error: at back already", p)
                                      [] ("", forwards(p)) )
                         in ("moveforward" concat k', p'))
  C[[moveback]] = λp. let (k', p') = isempty (p) → ("error: file is empty", p)
                                 [] ( at-first-record(p) → ("error: at front already", p) )
                                      [] ("", backwards(p))
                         in ("moveback" concat k', p')
  C[[insert R]] = λp. ("insert R", insert(R[[R]], p))
  C[[delete]] = λp. let (k', p') = isempty(p) → ("error: file is empty", p)
                                 [] ("", delete(p))
                         in ("delete" concat k', p')
```

```
                                 [] ("", delete(newfile))
        in ("delete" concat k', p'))
    = let (k', p') = ("error: file is empty", newfile)
        in ("delete" concat k', p')
    = ("delete" concat "error: file is empty", newfile)
    = ("delete error: file is empty", newfile)
```

The **S** function collects the log messages into a list. **S**[[**quit**]] builds the very end of this list. The equation for **S**[[C **cr** S]] deserves a bit of study. It says to:

1. Evaluate **C**[[C]]p to obtain the next log entry l' plus the updated open file p'.
2. Cons l' to the log list and pass p' onto **S**[[S]].
3. Evaluate **S**[[S]]p' to obtain the meaning of the remainder of the program, which is the rest of the log output plus the final version of the updated open file.

The two occurrences of **S**[[S]]p' may be a bit confusing. They do *not* mean to ''execute'' [[S]] twice— semantic definitions are functions, and the operational analogies are not always exact. The expression has the same meaning as:

let (l', p') = **C**[[C]]p in let (l'', p'') = **S**[[S]]p' in (l' *cons* l'', p'')

The **P** function is similar in spirit to **S**. (One last note: there is a bit of cheating in writing "edit I" as a token, because [[I]] is actually a piece of abstract syntax tree. A coercion function should be used to convert abstract syntax forms to string forms. This is of little importance

and is omitted.)

A small example shows how the log successfully collects terminal output. Let ⟦A⟧ be the name of a nonempty file in the file system s_0.

P⟦**edit** A **cr moveback cr delete cr quit**⟧s_0
= (″edit A″ *cons fst*(**S**⟦**moveback cr delete cr quit**⟧p_0),
update(⟦A⟧, *copyout*(*snd*(**S**⟦**moveback cr delete cr quit**⟧p_0), s_0))
where p_0 = *copyin*(*access*(⟦A⟧, s_0))

Already, the first line of terminal output is evident, and the remainder of the program can be simplified. After a number of simplifications, we obtain:

(″edit A″ *cons* ″moveback error: at front already″
cons fst(**S**⟦**delete cr quit**⟧p_0)),
update(⟦A⟧, *copyout*(*snd*(**S**⟦**delete cr quit**⟧p_0))))

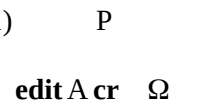
as the second command was incorrect. **S**⟦**delete cr quit**⟧p_0 simplifies to a pair (″delete quit″, p_1), for p_1 = *delete*(p_0), and the final result is:

(″edit A moveback error: at front already delete quit″,
update(⟦A⟧, *copyout*(p_1), s_0))

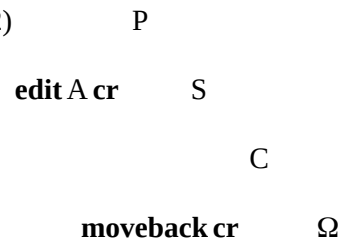
5.2.1 Interactive Input and Partial Syntax

A user of a file editor may validly complain that the above definition still isn't realistic enough, for interactive programs like text editors do not collect all their input into a single program before parsing and processing it. Instead, the input is processed incrementally— one line at a time. We might model incremental output by a series of abstract syntax trees. Consider again the sample program ⟦**edit** A **cr moveback cr delete cr quit**⟧. When the first line ⟦**edit** A **cr**⟧ is typed at the terminal, the file editor's parser can build an abstract syntax tree that looks like Diagram 5.1:

(5.1)



(5.2)



The parser knows that the first line of input is correct, but the remainder, the command sequence part, is unknown. It uses ⟦Ω⟧ to stand in place of the command sequence that follows. The tree in Diagram 5.1 can be pushed through the **P** function, giving **P**⟦**edit** A **cr** Ω⟧s_0 = (″**edit** A″ *cons fst*(**S**⟦Ω⟧p_0), *update*(⟦A⟧, *copyout*(*snd*(**S**⟦Ω⟧p_0), s_0)))

The processing has started, but the entire log and final file system are unknown.

When the user types the next command, the better-defined tree in Diagram 5.2 is built, and the meaning of the new tree is:

P⟦**edit** A **cr moveback cr** Ω⟧ =
(*"*edit A*"* *cons "*moveback error: at front already*"* *cons fst*(**S**⟦Ω⟧p_0),
update(⟦A⟧, *copyout*(*snd*(**S**⟦Ω⟧p_0)), s_0))

This denotation includes more information than the one for Diagram 5.1; it is ‘‘better defined.’’ The next tree is Diagram 5.3:

(5.3) P

edit A **cr** S

C S

moveback cr C

delete cr Ω

The corresponding semantics can be worked out in a similar fashion. An implementation strategy is suggested by the sequence: an implementation of the valuation function executes under the control of the editor’s parser. Whenever the parser obtains a line of input, it inserts it into a partial abstract syntax tree and calls the semantic processor, which continues its logging and file manipulation from the point where it left off, using the new piece of abstract syntax.

This idea can be formalized in an interesting way. Each of the abstract syntax trees was better defined than its predecessor. Let’s use the symbol $\sqsubseteq$ to describe this relationship. Thus, (5.1) $\sqsubseteq$ (5.2) $\sqsubseteq$ (5.3) $\sqsubseteq \cdots$ holds for the example. Similarly, we expect that **P**⟦(5.3)⟧s_0 contains more answer information than **P**⟦(5.2)⟧s_0, which itself has more information than **P**⟦(5.1)⟧s_0. If we say that the undefined value $\perp$ has the least answer information possible, we can define **S**⟦Ω⟧p=$\perp$ for all arguments p. The $\perp$ value stands for undetermined semantic information. Then we have that:

(*"*edit A*"* *cons* $\perp$, $\perp$)
$\sqsubseteq$ (*"*edit A*"* *cons "*moveback error: at front already*"* *cons* $\perp$, $\perp$)
$\sqsubseteq$ (*"*edit A*"* *cons "*moveback error: at front already*"* *cons "*delete*"* *cons* $\perp$, $\perp$)
$\sqsubseteq \cdots$

Each better-defined partial tree gives better-defined semantic information. We use these ideas in the next chapter for dealing with recursively defined functions.

5.3 A DYNAMICALLY TYPED LANGUAGE WITH INPUT AND OUTPUT

The third example language is an extension of the one in Section 5.1. Languages like SNOBOL allow variables to take on values from different data types during the course of evaluation. This provides flexibility to the user but requires that type checking be performed at runtime. The semantics of the language gives us insight into the type checking. Input and output are also included in the example.

Figure 5.5 gives the new semantic algebras needed for the language. The value domains that the language uses are the truth values *Tr* and the natural numbers *Nat*. Since these values can be assigned to identifiers, a domain:

$$Storable\text{-}value = Tr + Nat$$

is created. The + domain builder attaches a ''type tag'' to a value. The *Store* domain becomes:

$$Store = Id \to Storable\text{-}value$$

The type tags are stored with the truth values and numbers for later reference. Since storable values are used in arithmetic and logical expressions, type errors are possible, as in an attempt to add a truth value to a number. Thus, the values that expressions denote come from the domain:

Figure 5.5

V. Values that may be stored
Domain $v \in Storable\text{-}value = Tr + Nat$

VI. Values that expressions may denote
Domain $x \in Expressible\text{-}value = Storable\text{-}value + Errvalue$
where $Errvalue = Unit$
Operations

$check\text{-}expr : (Store \to Expressible\text{-}value) \times$
$(Storable\text{-}value \to Store \to Expressible\text{-}value) \to (Store \to Expressible\text{-}value)$
$f_1\ check\text{-}expr\ f_2 = \lambda s.$ cases $(f_1\ s)$ of
is*Storable-value*$(v) \to (f_2\ v\ s)$
[] is*Errvalue*$() \to$ in*Errvalue*()
end

VII. Input buffer
Domain $i \in Input = Expressible\text{-}value^*$
Operations

$get\text{-}value : Input \to (Expressible\text{-}value \times Input)$
$get\text{-}value = \lambda i.\ null\ i \to (\text{in}Errvalue(), i)$ [] $(hd\ i,\ tl\ i)$

Figure 5.5 (continued)

VIII. Output buffer

Domain $o \in Output = (Storable\text{-}value + String)^*$

Operations

$empty : Output$

$empty = nil$

$put\text{-}value : Storable\text{-}value \times Output \to Output$

$put\text{-}value = \lambda(v,o).\ \text{in}Storable\text{-}value(v)\ cons\ o$

$put\text{-}message : String \times Output \to Output$

$put\text{-}message = \lambda(t,o).\ \text{in}String(t)\ cons\ o$

IX. Store

Domain $s \in Store = \text{Id} \to Storable\text{-}value$

Operations

$newstore : Store$

$access : Id \to Store \to Storable\text{-}value$

$update : Id \to Storable\text{-}value \to Store \to Store$

X. Program State

Domain $a \in State = Store \times Input \times Output$

XI. Post program state

Domain $z \in Post\text{-}state = OK + Err$

where $OK = State$

and $Err = State$

Operations

$check\text{-}result : (Store \to Expressible\text{-}value) \times (Storable\text{-}value \to State \to Post\text{-}state_\perp) \to (State \to Post\text{-}state_\perp)$

$f\ check\text{-}result\ g = \lambda(s,i,o).\ \text{cases}\ (f\ s)\ \text{of}$
$\quad \text{is}Storable\text{-}value(v) \to (g\ v\ (s,i,o))$
$\quad [] \ \text{is}Errvalue() \to \text{in}Err(s,\ i,\ put\text{-}message(\text{"type error"},\ o))\ \text{end}$

$check\text{-}cmd : (State \to Post\text{-}state_\perp) \times (State \to Post\text{-}state_\perp) \to (State \to Post\text{-}state_\perp)$

$h_1\ check\text{-}cmd\ h_2 = \lambda a.\ \text{let}\ z = (h_1\ a)\ \text{in cases}\ z\ \text{of}$
$\quad \text{is}OK(s,i,o) \to h_2(s,i,o)$
$\quad [] \ \text{is}Err(s,i,o) \to z\ \text{end}$

Expressible-value = *Storable-value* + *Errvalue*

where the domain *Errvalue* = *Unit* is used to denote the result of a type error.

Of interest is the program state, which is a triple of the store and the input and output buffers. The *Post-state* domain is used to signal when an evaluation is completed successfully and when a type error occurs. The tag attached to the state is utilized by the *checkcmd* operation. This operation is the sequencing operation for the language and is represented in infix form. The expression (**C**⟦C_1⟧ *check-cmd* **C**⟦C_2⟧) does the following:

1. It gives the current state *a* to **C**⟦C_1⟧, producing a post-state *z* = **C**⟦C_1⟧*a*.
2. If *z* is a proper state a', and then, if the state component is *OK*, it produces **C**⟦C_2⟧a_1. If *z* is erroneous, **C**⟦C_2⟧ is ignored (it is ''branched over''), and *z* is the result.

A similar sequencing operation, *check-result*, sequences an expression with a command. For example, in an assignment ⟦I:=E⟧, ⟦E⟧'s value must be determined before a store update can occur. Since ⟦E⟧'s evaluation may cause a type error, the error must be detected before the update is attempted. Operation *check-result* performs this action. Finally, *check-expr* performs error trapping at the expression level.

Figure 5.6 shows the valuation functions for the language.

You are encouraged to write several programs in the language and derive their denotations. Notice how the algebra operations abort normal evaluation when type errors occur. The intuition behind the operations is that they represent low-level (even hardware-level) fault detection and branching mechanisms. When a fault is detected, the usual machine action is a single branch out of the program. The operations defined here can only ''branch'' out of a subpart of the function expression, but since all type errors are propagated, these little branches chain together to form a branch out of the entire program. The implementor of the language would take note of this property and produce full jumps on error detection. Similarly, the in*OK* and in*Err* tags would not be physically implemented, as any running program has an *OK* state, and any error branch causes a change to the *Err* state.

5.4 ALTERING THE PROPERTIES OF STORES

The uses of the store argument in this chapter maintain properties 1-3 noted in the introduction to this chapter. These properties limit the use of stores. Of course, the properties are limiting in the sense that they describe typical features of a store in a sequential programming language. It is instructive to relax each of restrictions 1, 3, and 2 in turn and see what character of programming languages result.

5.4.1 Delayed Evaluation

Call-by-value (argument first) simplification is the safe method for rewriting operator, argument combinations when strict functions are used. This point is important, for it suggests that an implementation of the strict function needs an evaluated argument to proceed. Similarly,

Figure 5.6

Abstract syntax:

P∈ Program
C∈ Command
E∈ Expression
I∈ Id
N∈ Numeral

P ::= C.
C ::= C_1;C_2 | I:=E | **if** E **then** C_1 **else** C_2 | **read** I | **write** E | **diverge**
E ::= E_1+E_2 | E_1=E_2 | ¬E | (E) | I | N | **true**

Semantic algebras:

I. Truth values (defined in Figure 5.1)

II. Natural numbers (defined in Figure 5.1)

III. Identifiers (defined in Figure 5.1)

IV. Character strings (defined in Example 3.5 of Chapter 3)

V. - XI. (defined in Figure 5.5)

Valuation functions:

P: Program→ *Store*→ *Input*→ *Post-state*$_\perp$
P⟦C.⟧ = λ*s*.λ*i*. **C**⟦C⟧ (*s*, *i*, *empty*)

C: Command→ *State*→ *Post-state*$_\perp$
C⟦C_1;C_2⟧ = **C**⟦C_1⟧ *check-cmd* **C**⟦C_2⟧
C⟦I:=E⟧ = **E**⟦E⟧ *check-result* (λ*v*.λ(*s*,*i*,*o*). in*OK*((*update*⟦I⟧ *v s*), *i*, *o*))
C⟦**if** E **then** C_1 **else** C_2⟧ = **E**⟦E⟧ *check-result*
(λ*v*.λ(*s*,*i*,*o*). cases *v* of
is*Tr*(*t*)→ (*t*→ **C**⟦C_1⟧ [] **C**⟦C_2⟧)(*s*,*i*,*o*)
[] is*Nat*(*n*)→ in*Err*(*s*,*i*, *put-message*("bad test", *o*)) end)
C⟦**read** I⟧ = λ(*s*,*i*,*o*). let (*x*,*i*′) = *get-value*(*i*) in
cases *x* of
is*Storable-value*(*v*) → in*OK*((*update*⟦I⟧ *v s*), *i*′, *o*)
[] is*Errvalue*() → in*Err*(*s*, *i*′, *put-message*("bad input", *o*)) end
C⟦**write** E⟧ = **E**⟦E⟧ *check-result* (λ*v*.λ(*s*,*i*,*o*). in*OK*(*s*, *i*, *put-value*(*v*,*o*)))
C⟦**diverge**⟧ = λ*a*. ⊥

Figure 5.6 (continued)

```
E: Expression→ Store→ Expressible-value
  E⟦E1+E2⟧ = E⟦E1⟧ check-expr
        (λv. cases v of
              isTr(t)→ λs. inErrvalue()
              [] isNat(n)→ E⟦E2⟧ check-expr
                    (λv'.λs. cases v' of
                          isTr(t')→ inErrvalue()
                          [] isNat(n')→ inStorable-value(inNat(n plus n')) end)
              end)
  E⟦E1=E2⟧ = ''similar to above equation''
  E⟦¬E⟧ = E⟦E⟧ check-expr
        (λv.λs. cases v of
              isTr(t)→ inStorable-value(inTr(not t))
              [] isNat(n)→ inErrvalue() end)
  E⟦(E)⟧ = E⟦E⟧
  E⟦I⟧ = λs. inStorable-value(access ⟦I⟧ s)
  E⟦N⟧ = λs. inStorable-value(inNat(N⟦N⟧))
  E⟦true⟧ = λs. inStorable-value(inTr(true))
N:Numeral→ Nat (omitted)
```

call-by-name (argument last) simplification is the safe method for handling arguments to nonstrict functions. Here is an example: consider the nonstrict function $f = (\lambda x.\, zero)$ of domain $Nat_\perp \to Nat_\perp$. If f is given an argument e whose meaning is $\perp$, then $f(e)$ is *zero*. Argument e's simplification may require an infinite number of steps, for it represents a nonterminating evaluation. Clearly, e should not be simplified if given to a nonstrict f.

The *Store*-based operations use only proper arguments and a store can only hold values that are proper. Let's consider how stores might operate with improper values. First, say that expression evaluation can produce both proper and improper values. Alter the *Store* domain to be $Store = Id \to Nat_\perp$. Now improper values may be stored. Next, adjust the *update* operation to be: $update : Id \to Nat_\perp \to Store \to Store$, $update = \lambda i.\lambda n.\lambda s.\, [\, i \mapsto n\,]s$. An assignment statement uses *update* to store the value of an expression ⟦E⟧ into the store. If ⟦E⟧ represents a ''loop forever'' situation, then $\mathbf{E}[\![E]\!]s = \perp$. But, since *update* is nonstrict in its second argument, (*update* ⟦I⟧ (**E**⟦E⟧*s*) *s*) is defined. From the operational viewpoint, unevaluated or partially evaluated expressions may be stored into *s*. The form **E**⟦E⟧*s* need not be evaluated until it is used; the arrangement is called *delayed* (or *lazy*) *evaluation.* Delayed evaluation provides the advantage that the only expressions evaluated are the ones that are actually needed for

computing answers. But, once **E**⟦E⟧'s value is needed, it must be determined with respect to the store that was active when ⟦E⟧ was saved. To understand this point, consider this code:

begin
    X:=0;
    Y:=X+1;
    X:=4
resultis Y

where the block construct is defined as:

K: Block→ $Store_{\perp} \rightarrow Nat_{\perp}$
K⟦**begin** C **resultis** E⟧ = $\underline{\lambda}s.$ **E**⟦E⟧ (**C**⟦C⟧s)

(Note: **E** now has functionality **E** : Expression → $Store_{\perp} \rightarrow Nat_{\perp}$, and it is strict in its store argument.) At the final line of the example, the value of ⟦Y⟧ must be determined. The semantics of the example, with some proper store s_0, is:

K⟦**begin** X:=0; Y:=X+1; X:=4 **resultis** Y⟧s_0
= **E**⟦Y⟧ (**C**⟦X:=0; Y:=X+1; X:=4⟧s_0)
= **E**⟦Y⟧ (**C**⟦Y:=X+1; X:=4⟧ (**C**⟦X:=0⟧s_0))
= **E**⟦Y⟧ (**C**⟦Y:=X+1; X:=4⟧ (*update*⟦X⟧ (**E**⟦0⟧s_0) s_0))

At this point, (**E**⟦0⟧s_0) need not be simplified; a new, proper store, s_1 = (*update*⟦X⟧ **E**⟦0⟧s_0 s_0) is defined regardless. Continuing through the other two commands, we obtain:

s_3 = *update*⟦X⟧ (**E**⟦4⟧s_2) s_2
    where s_2 = *update*⟦Y⟧ (**E**⟦X+1⟧s_1) s_1

and the meaning of the block is:

E⟦Y⟧s_3 = *access*⟦Y⟧ s_3
= **E**⟦X+1⟧s_1
= **E**⟦X⟧s_1 *plus one*
= (*access*⟦X⟧ s_1) *plus one*)
= **E**⟦0⟧s_0 *plus one*
= *zero plus one* = *one*

The old version of the store, version s_1, must be retained to obtain the proper value for ⟦X⟧ in ⟦X+1⟧. If s_3 was used instead, the answer would have been the incorrect *five*.

Delayed evaluation can be carried up to the command level by making the **C**, **E**, and **K** functions nonstrict in their store arguments. The surprising result is that only those commands that have an effect on the output of a program need be evaluated. Convert all strict abstractions ($\underline{\lambda}s.\ e$) in the equations for **C** in Figure 5.2 to the nonstrict forms ($\lambda s.\ e$). Redefine *access* and *update* to be:

$access$: Identifier $\rightarrow Store_{\perp} \rightarrow Nat_{\perp}$
$\quad access = \lambda i.\underline{\lambda} s.\ s(i)$
$update$: Identifier $\rightarrow Nat_{\perp} \rightarrow Store_{\perp} \rightarrow Store_{\perp}$
$\quad update = \lambda i.\lambda m.\lambda p.\ (\lambda i'.\ i'\ equals\ i \rightarrow m\ [\!]\ (access\ i'\ p))$

Then, regardless of the input store s, the program:

begin
  X:=0;
  diverge;
  X:=2
resultis X+1

has the value *three*! This is because $\mathbf{C}[\![\text{X:=0; }\mathbf{diverge}]\!]s = \perp$, and:

$\mathbf{E}[\![\text{X+1}]\!](\mathbf{C}[\![\text{X:=2}]\!]\perp)$
$= \mathbf{E}[\![\text{X+1}]\!](update[\![\text{X}]\!](\mathbf{E}[\![2]\!]\perp)\perp)$, as **C** is nonstrict
$= \mathbf{E}[\![\text{X+1}]\!]([\,[\![\text{X}]\!] \mapsto \mathbf{E}[\![2]\!]\perp\,]\perp)$, as *update* is nonstrict
$= \mathbf{E}[\![\text{X}]\!]([\,[\![\text{X}]\!] \mapsto \mathbf{E}[\![2]\!]\perp\,]\perp)$ *plus one*
$= (access[\![\text{X}]\!]([\,[\![\text{X}]\!] \mapsto \mathbf{E}[\![2]\!]\perp\,]\perp))$ *plus one*
$= \mathbf{E}[\![2]\!]\perp$ *plus one*
$=$ *two plus one*, as **E** is nonstrict
$=$ *three*

The derivation suggests that only the last command in the block need be evaluated to obtain the answer. Of course, this goes against the normal left-to-right, top-to-bottom sequentiality of command evaluation, so the nonstrict handling of stores requires a new implementation strategy.

5.4.2 Retaining Multiple Stores

Relaxing the strictness condition upon stores means that multiple values of stores must be present in an evaluation. Must an implementation of any of the languages defined earlier in this chapter use multiple stores? At first glance, the definition of addition:

$$\mathbf{E}[\![\text{E}_1\text{+E}_2]\!] = \lambda s.\ \mathbf{E}[\![\text{E}_1]\!]s\ plus\ \mathbf{E}[\![\text{E}_2]\!]s$$

apparently does need two copies of the store to evaluate. Actually, the format is a bit deceiving. An implementation of this clause need only retain one copy of the store s because both $\mathbf{E}[\![\text{E}_1]\!]$ and $\mathbf{E}[\![\text{E}_2]\!]$ use s in a ''read only'' mode. Since s is not updated by either, the equation should be interpreted as saying that the order of evaluation of the two operands to the addition is unimportant. They may even be evaluated in parallel. The obvious implementation of the store is a global variable that both operands may access.

This situation changes when side effects occur within expression evaluation. If we add

the block construct to the Expression syntax domain and define its semantics to be:

$$\mathbf{E}[\![\textbf{begin } \mathrm{C} \textbf{ resultis } \mathrm{E}]\!] = \underline{\lambda} s.\ \text{let } s' = \mathbf{C}[\![\mathrm{C}]\!]s \text{ in } \mathbf{E}[\![\mathrm{E}]\!]s'$$

then expressions are no longer ''read only'' objects. An implementation faithful to the semantic equation must allow an expression to own a local copy of store. The local store and its values disappear upon completion of expression evaluation. To see this, you should perform the simplification of $\mathbf{C}[\![\mathrm{X}:=(\textbf{begin } \mathrm{Y}:=\mathrm{Y}+1 \textbf{ resultis } \mathrm{Y})+\mathrm{Y}]\!]$. The incrementation of $[\![\mathrm{Y}]\!]$ in the left operand is unknown to the right operand. Further, the store that gets the new value of $[\![\mathrm{X}]\!]$ is exactly the one that existed prior to the right-hand side's evaluation.

The more conventional method of integrating expression-level updates into a language forces any local update to remain in the global store and thus affect later evaluation. A more conventional semantics for the block construct is:

$$\mathbf{K}[\![\textbf{begin } \mathrm{C} \textbf{ resultis } \mathrm{E}]\!] = \underline{\lambda} s.\ \text{let } s' = \mathbf{C}[\![\mathrm{C}]\!]s \text{ in } (\mathbf{E}[\![\mathrm{E}]\!]s', s')$$

The expressible value and the updated store form a pair that is the result of the block.

5.4.3 Noncommunicating Commands

The form of communication that a store facilitates is the building up of side effects that lead to some final value. The purpose of a command is to advance a computation a bit further by drawing upon the values left in the store by previous commands. When a command is no longer allowed to draw upon the values, the communication breaks down, and the language no longer has a sequential flavor.

Let's consider an example that makes use of multiple stores. Assume there exists some domain D with an operation $combine : D \times D \to D$. If *combine* builds a ''higher-quality'' D-value from its two D-valued arguments, a useful store-based, noncommunicating semantics might read:

$$\begin{aligned}
&\text{Domain } s \in Store = Id \to D\\
&\mathbf{C}\text{: Command} \to Store_{\perp} \to Store_{\perp}\\
&\mathbf{C}[\![\mathrm{C}_1;\mathrm{C}_2]\!] = \underline{\lambda} s.\ join\ (\mathbf{C}[\![\mathrm{C}_1]\!]s)\ (\mathbf{C}[\![\mathrm{C}_2]\!]s)\\
&\quad \text{where } join : Store_{\perp} \to Store_{\perp} \to Store_{\perp}\\
&\quad\quad join = \underline{\lambda} s_1.\underline{\lambda} s_2.\ (\lambda i.\ s_1(i)\ combine\ s_2(i))
\end{aligned}$$

These clauses suggest parallel but noninterfering execution of commands. Computing is divided between $[\![\mathrm{C}_1]\!]$ and $[\![\mathrm{C}_2]\!]$ and the partial results are joined using *combine*. This is a nontraditional use of parallelism on stores; the traditional form of parallelism allows interference and uses the single-store model. Nonetheless, the above example is interesting because it suggests that noncommunicating commands can work together to build answers rather than deleting each other's updates.

SUGGESTED READINGS

Semantics of the store and assignment: Barron 1977; Donohue 1977; Friedman et al. 1984; Landin 1965; Strachey 1966, 1968
Interactive systems: Bjørner and Jones 1982; Cleaveland 1980
Dynamic typing: Tennent 1973
Delayed evaluation: Augustsson 1984; Friedman & Wise 1976; Henderson 1980; Henderson & Morris 1976

EXERCISES

1. Determine the denotations of the following programs in $Nat_\perp$ when they are used with the input data value *one*:

   a. **P**⟦Z:=A.⟧
   b. **P**⟦(**if** A=0 **then diverge else** Y:=A+1);Z:=Y.⟧
   c. **P**⟦**diverge**; Z:=0.⟧

2. Determine the denotations of the programs in the previous exercise without any input; that is, give their meanings in the domain $Nat \to Nat_\perp$.

3. Give an example of a program whose semantics with respect to Figure 5.2, is the denotation ($\lambda n.\, one$). Does an algorithmic method exist for listing all the programs with exactly this denotation?

4. Show that the following properties hold with respect to the semantic definition of Figure 5.2:

   a. **P**⟦Z:=0; **if** A=0 **then** Z:=A.⟧ = **P**⟦Z:=0.⟧
   b. For any C∈ Command, **C**⟦**diverge**; C⟧ = **C**⟦**diverge**⟧
   c. For all E_1, E_2∈ Expression, **E**⟦E_1+E_2⟧ = **E**⟦E_2+E_1⟧
   d. For any B∈ Boolean-expr, C_1, C_2∈ Command,
      C⟦**if** B **then** C_1 **else** C_2⟧ = **C**⟦**if** ¬B **then** C_2 **else** C_1⟧.
   e. There exist some B∈ Boolean-expr and C_1, C_2∈ Command such that
      C⟦**if** B **then** C_1; **if** ¬B **then** C_2⟧ ≠ **C**⟦**if** B **then** C_1 **else** C_2⟧

   (Hint: many of the proofs will rely on the extensionality of functions.)

5. a. Using structural induction, prove the following: for every E∈ Expression in the language of Figure 5.2, for any I∈ Identifier, *E'*∈ Expression, and $s \in Store$, **E**⟦[E′/I]E⟧s = **E**⟦E⟧(*update* ⟦I⟧ **E**⟦E′⟧s s).
   b. Use the result of part a to prove: for every B∈ Boolean-expr in the language of Figure 5.2, for every I∈ Identifier, *E'*∈ Expression, and $s \in Store$, **B**⟦[E′/I]B⟧s = **B**⟦B⟧(*update* ⟦I⟧ **E**⟦E′⟧s s).

6. Say that the *Store* algebra in Figure 5.1 is redefined so that the domain is $s \in Store' = (Id \times Nat)^*$.

   a. Define the operations *newstore', access',* and *update'* to operate upon the new domain. (For this exercise, you are allowed to use a recursive definition for *access'*. The definition must satisfy the properties stated in the solution to Exercise 14, part b, of Chapter 3.) Must the semantic equations in Figure 5.2 be adjusted to work with the new algebra?

   b. Prove that the definitions created in part a satisfy the properties: for all $i \in Id$, $n \in Nat$, and $s \in Store'$:

   $$access'\ i\ newstore' = zero$$
   $$access'\ i\ (update'\ i\ n\ s) = n$$
   $$access'\ i\ (update'\ j\ n\ s) = (access'\ i\ s), \text{ for } j \neq i$$

   How do these proofs relate the new *Store* algebra to the original? Try to define a notion of ''equivalence of definitions'' for the class of all *Store* algebras.

7. Augment the Command syntax domain in Figure 5.2 with a **swap** command:

   C ::= $\cdots$ | **swap** I_1, I_2

   The action of **swap** is to interchange the values of its two identifier variables. Define the semantic equation for **swap** and prove that the following property holds for any $J \in Id$ and $s \in Store$: $\mathbf{C}[\![\textbf{swap}\ J, J]\!]s = s$. (Hint: appeal to the extensionality of store functions.)

8. a. Consider the addition of a Pascal-like **cases** command to the language of Figure 5.2. The syntax goes as follows:

   C∈ Command
   G∈ Guard
   E∈ Expression
   C ::= $\cdots$ | **case** E **of** G **end**
   G ::= N:C; G | N:C

   Define the semantic equation for $\mathbf{C}[\![\textbf{case}\ E\ \textbf{of}\ G\ \textbf{end}]\!]$ and the equations for the valuation function $\mathbf{G}$: Guard $\rightarrow$ $(Nat \times Store) \rightarrow Store_\perp$. List the design decisions that must be made.

   b. Repeat part a with the rule G ::= N:C | G_1; G_2

9. Say that the command $[\![\textbf{test}\ E\ \textbf{on}\ C]\!]$ is proposed as an extension to the langauge of Figure 5.2. The semantics is:

   $$\mathbf{C}[\![\textbf{test}\ E\ \textbf{on}\ C]\!] = \underline{\lambda} s.\ \text{let } s' = \mathbf{C}[\![C]\!]s \text{ in } \mathbf{E}[\![E]\!]s'\ equals\ zero \rightarrow s', [\,]\ s$$

   What problems do you see with implementing this construct on a conventional machine?

10. Someone proposes a version of ''parallel assignment'' with semantics:

    $$\mathbf{C}[\![I_1, I_2 := E_1, E_2]\!] = \underline{\lambda} s.\ \text{let } s' = (update[\![I_1]\!]\ \mathbf{E}[\![E_1]\!]s\ s)$$

in *update* $[\![I_2]\!]$ $\mathbf{E}[\![E_2]\!]s, s$

Show, via a counterexample, that the semantics does not define a true parallel assignment. Propose an improvement. What is the denotation of $[\![J, J := 0, 1]\!]$ in your semantics?

11. In a LUCID-like language, a family of parallel assignments are performed in a construct known as a *block.* The syntax of a block B is:

    B ::= **begin** A **end**
    A ::= I_{new}:=E | A_1 § A_2

    The block is further restricted so that all identifiers on the left hand sides of assignments in a block must be distinct. Define the semantics of the block construct.

12. Add the **diverge** construction to the syntax of Expression in Figure 5.2 and say that $\mathbf{E}[\![\mathbf{diverge}]\!] = \lambda s.\ \bot$. How does this addition impact:

    a. The functionalities and semantic equations for **C**, **E**, and **B**?
    b. The definition and use of the operations *update, plus, equals,* and *not*? What is your opinion about allowing the possibility of nontermination in expression evaluation? What general purpose imperative languages do you know of that guarantee termination of expression evaluation?

13. The document defining the semantics of Pascal claims that the order of evaluation of operands in an (arithmetic) expression is left unspecified; that is, a machine may evaluate the operands in whatever order it pleases. Is this concept expressed in the semantics of expressions in Figure 5.2? However, recall that Pascal expressions may contain side effects. Let's study this situation by adding the construct $[\![C$ **in** $E]\!]$. Its evaluation first evaluates $[\![C]\!]$ and then evaluates $[\![E]\!]$ using the store that was updated by $[\![C]\!]$. The store (with the updates) is passed on for later use. Define $\mathbf{E}[\![C$ **in** $E]\!]$. How must the functionality of **E** change to accommodate the new construct? Rewrite all the other semantic equations for **E** as needed. What order of evaluation of operands does your semantics describe? Is it possible to specify a truly nondeterminate order of evaluation?

14. For some defined store s_0, give the denotations of each of the following file editor programs, using the semantics in Figure 5.4:

    a. $\mathbf{P}[\![$**edit** A **cr newfile cr insert** R_0 **cr insert** R_1 **quit**$]\!]s_0$. Call the result (log_1, s_1).
    b. $\mathbf{P}[\![$**edit** A **cr moveforward cr delete cr insert** R_2 **quit**$]\!]s_1$, where s_1 is from part a. Call the new result (log_2, s_2).
    c. $\mathbf{P}[\![$**edit** A **cr insert** R_3 *cr* **quit**$]\!]s_2$, where s_2 is from part b.

15. Redo part a of the previous question in the style described in Section 5.2.1, showing the partial syntax trees and the partial denotations produced at each step.

16. Extend the file editor of Figure 5.4 to be a text editor: define the internal structure of the

Record semantic domain in Figure 5.3 and devise operations for manipulating the words in a record. Augment the syntax of the language so that a user may do manipulations on the words within individual records.

17. Design a programming language for performing character string manipulation. The language should support fundamental operations for pattern matching and string manipulation and possess assignment and control structure constructs for imperative programming. Define the semantic algebras first and then define the abstract syntax and valuation functions.

18. Design a semantics for the grocery store data base language that you defined in Exercise 6 of Chapter 1. What problems arise because the abstract syntax was defined before the semantic algebras? What changes would you make to the language's syntax after this exercise?

19. In the example in Section 5.3, the *Storable-value* domain is a subdomain of the *Expressible-value* domain; that is, every storable value is expressible. What problems arise when this isn't the case? What problems/situations arise when an expressible value isn't storable? Give examples.

20. In the language of Figure 5.6, what is $\mathbf{P}[\![\textbf{write}\ 2; \textbf{diverge}.]\!]$? Is this a satisfactory denotation for the program? If not, suggest some revisions to the semantics.

21. Alter the semantics of the language of Figure 5.6 so that an expressible value error causes an error message to be placed into the output buffer immediately (rather than letting the command in which the expressible value is embedded report the message later).

22. Extend the *Storable-value* algebra of Figure 5.5 so that arithmetic can be performed on the (numeric portion of) storable values. In particular, define operations:

$$plus_{\mathbf{s}} : \textit{Storable-value} \times \textit{Storable-value} \to \textit{Expressible-value}$$
$$not_{\mathbf{s}} : \textit{Storable-value} \to \textit{Expressible-value}$$
$$equals_{\mathbf{s}} : \textit{Storable-value} \times \textit{Storable-value} \to \textit{Expressible-value}$$

so that the equations in the **E** valuation function can be written more simply, e.g.,

$$\mathbf{E}[\![\mathrm{E}_1 + \mathrm{E}_2]\!] = \mathbf{E}[\![\mathrm{E}_1]\!]s\ \textit{check-expr}\ (\lambda v_1.\ \mathbf{E}[\![\mathrm{E}_2]\!]s\ \textit{check-expr}\ (\lambda v_2.\ v_1\ \textit{plus}_{\mathbf{s}}\ v_2))$$

Rewrite the other equations of **E** in this fashion. How would the new versions of the storable value operations be implemented on a computer?

23. Alter the semantics of the language of Figure 5.6 so that a variable retains the type of the first identifier that is assigned to it.

24. a. Alter the *Store* algebra in Figure 5.5 so that:

$$\textit{Store} = \textit{Index} \to \textit{Storable-value}^*$$

where $Index = Id + Input + Output$
$Input = Unit$
$Output = Unit$

that is, the input and output buffers are kept in the store and indexed by tags. Define the appropriate operations. Do the semantic equations require alterations?

b. Take advantage of the new definition of storage by mapping a variable to a *history* of all its updates that have occurred since the program has been running.

25. Remove the command ⟦**read** I⟧ from the language of Figure 5.6 and place the construct ⟦**read**⟧ into the syntax of expressions.

a. Give the semantic equation for **E**⟦**read**⟧.
b. Prove that **C**⟦**read** I⟧ = **C**⟦I:= **read**⟧.
c. What are the pragmatic advantages and disadvantages of the new construct?

26. Suppose that the *Store* domain is defined to be $Store = Id \to (Store \to Nat)$ and the semantic equation for assignment is:

$\mathbf{C}[\![\text{I:=E}]\!] = \underline{\lambda} s.\ update\ [\![\text{I}]\!]\ (\mathbf{E}[\![\text{E}]\!])\ s$

a. Define the semantic equations for the **E** valuation function.
b. How does this view of expression evaluation differ from that given in Figures 5.1 and 5.2? How is the new version like a macroprocessor? How is it different?

27. If you are familiar with data flow and demand-driven languages, comment on the resemblance of the nonstrict version of the **C** valuation function in Section 5.4.1 to these forms of computation.

28. Say that a vendor has asked you to design a simple, general purpose, imperative programming language. The language will include concepts of *expression* and *command*. Commands update the store; expressions do not. The *control structures* for commands include sequencing and conditional choice.

a. What questions should you ask the vendor about the language's design? Which design decisions should you make without consulting the vendor first?
b. Say that you decide to use denotational semantics to define the semantics of the language. How does its use direct and restrict your view of:

i. What the store should be?
ii. How stores are accessed and updated?
iii. What the order of evaluation of command and expression subparts should be?
iv. How the control structures order command evaluation?

29. Programming language design has traditionally worked from a ''bottom up'' perspective; that is, given a physical computer, a machine language is defined for giving instructions to the computer. Then, a second language is designed that is ''higher level'' (more concise or easier for humans to use) than the first, and a translator program is written to

translate from the second language to the first.

Why does this approach limit our view as to what a programming language should be? How might we break out of this approach by using denotational semantics to design new languages? What biases do we acquire when we use denotational semantics?