Most serious programming languages combine imperative aspects, which describe computation in terms of state-transformation operations such as assignment, and functional or applicative aspects, which describe computation in terms of the definition and application of functions or procedures. To gain a solid understanding, however, it is best to begin by considering each of these aspects in isolation, and to postpone the complications that arise from their interactions.

Thus, beginning in this chapter and continuing through Chapter 7 (nondeterminism) and Chapters 8 and 9 (concurrency), we will limit ourselves to purely imperative languages. Then, beginning in Chapter 10, we will turn to purely functional languages. Languages that combine imperative and functional aspects will be considered in Chapter 13 (Iswim-like languages) and Chapter 19 (Algol-like languages).

In this chapter, we consider a simple imperative language that is built out of assignment commands, sequential composition, conditionals (i.e. if commands), while commands, and (in Section 2.5) variable declarations. We will use this language to illustrate the basic concept of a domain, to demonstrate the properties of binding in imperative languages, and, in the next chapter, to explore formalisms for specifying and proving imperative program behavior. In later chapters we will explore extensions to this language and other approaches to describing its semantics.

At an intuitive level, the simple imperative language is so much a part of every programmer’s background that it will hold few surprises for typical readers. As in Chapter 1, we have chosen for clarity’s sake to introduce novel concepts such as domains in a familiar context. More surprising languages will come after we have sharpened our tools for specifying them.

2.1 Syntax

Our version of the simple imperative language has three types of phrases: integer expressions, boolean expressions, and commands. Its abstract syntax is described
by the following abstract grammar:

\[
\begin{align*}
\langle \text{intexp} \rangle & ::= 0 | 1 | 2 | \cdots \\
& \quad | \langle \text{var} \rangle | - \langle \text{intexp} \rangle | \langle \text{intexp} \rangle + \langle \text{intexp} \rangle | \langle \text{intexp} \rangle - \langle \text{intexp} \rangle \\
& \quad | \langle \text{intexp} \rangle \times \langle \text{intexp} \rangle | \langle \text{intexp} \rangle \div \langle \text{intexp} \rangle | \langle \text{intexp} \rangle \text{ rem } \langle \text{intexp} \rangle \\
\langle \text{boolexp} \rangle & ::= \text{true} | \text{false} \\
& \quad | \langle \text{intexp} \rangle = \langle \text{intexp} \rangle | \langle \text{intexp} \rangle \neq \langle \text{intexp} \rangle | \langle \text{intexp} \rangle < \langle \text{intexp} \rangle \\
& \quad | \langle \text{intexp} \rangle \leq \langle \text{intexp} \rangle | \langle \text{intexp} \rangle > \langle \text{intexp} \rangle | \langle \text{intexp} \rangle \geq \langle \text{intexp} \rangle \\
& \quad | \neg \langle \text{boolexp} \rangle | \langle \text{boolexp} \rangle \land \langle \text{boolexp} \rangle | \langle \text{boolexp} \rangle \lor \langle \text{boolexp} \rangle \\
& \quad | \langle \text{boolexp} \rangle \Rightarrow \langle \text{boolexp} \rangle | \langle \text{boolexp} \rangle \Leftrightarrow \langle \text{boolexp} \rangle \\
\langle \text{comm} \rangle & ::= \langle \text{var} \rangle := \langle \text{intexp} \rangle | \text{skip} | \langle \text{comm} \rangle ; \langle \text{comm} \rangle \\
& \quad | \text{if } \langle \text{boolexp} \rangle \text{ then } \langle \text{comm} \rangle \text{ else } \langle \text{comm} \rangle \\
& \quad | \text{while } \langle \text{boolexp} \rangle \text{ do } \langle \text{comm} \rangle
\end{align*}
\]

Anticipating that, in the next chapter, we will use predicate logic to specify programs, we have chosen expressions that are as close as possible to predicate logic: Integer expressions are exactly the same, and boolean expressions are the same as assertions except for the omission of quantifiers (for the obvious reason that they are noncomputable).

We will parenthesize expressions in the same way as in the previous chapter, while giving the assignment and sequencing operators separate levels of precedence that are lower than any of the arithmetic or logical operators. Thus the precedence list is

\[
(\times \div \text{ rem}) (-\text{unary } + -\text{binary})(= \neq \leq > \geq) \neg \land \lor \Rightarrow \Leftrightarrow := ;
\]

In the commands if \( b \) then \( c_0 \) else \( c_1 \) and while \( b \) do \( c_1 \), the subphrase \( c_1 \) will extend to the first stopping symbol or the end of the enclosing phrase. (Note that ";" is a stopping symbol, but "=" is not.)

Strictly speaking, we should add the assumption that the sequencing operator ";" is left associative. In practice, however, this is unimportant, since we will always give this operator an associative semantics where \( (c_0 ; c_1) ; c_2 \) and \( c_0 ; (c_1 ; c_2) \) have the same meaning. (Associativity is irrelevant for "="; since neither \( x_0 := (x_1 := x_2) \) nor \( (x_0 := x_1) := x_2 \) satisfies our abstract syntax.)

Somewhat simplistically, we assume that all variables take on integer values; in particular, there are no boolean variables. We will consider languages with more than one type of variable when we introduce type systems in Chapter 15; for the present, however, multiple types of variables would only complicate our definitions and obscure more important concepts.
2.2 Denotational Semantics

The semantics of integer expressions is the same as in the previous chapter, and, except for the omission of quantifiers, the semantics of boolean expressions is the same as that of assertions in the previous chapter. Thus the semantic functions

\[ [-]_{\text{intexp}} \in \langle \text{intexp} \rangle \rightarrow \Sigma \rightarrow \mathbb{Z} \]
\[ [-]_{\text{boolexp}} \in \langle \text{boolexp} \rangle \rightarrow \Sigma \rightarrow \mathbb{B} \]

are defined by semantic equations (1.3) to (1.10) in Section 1.2, with the replacement of \([-]_{\text{assert}}\) by \([-]_{\text{boolexp}}\).

(As in the previous chapter, expressions always terminate without an error stop. In particular, division by zero must produce some integer result. We will discuss this difficulty in Section 2.7.)

The semantics of commands, however, is quite different. Since the behavior of a command is essentially to transform the state of a computation from an initial state to a final state, one would expect the meaning of a command to be a state-transformation function from \(\Sigma\) to \(\Sigma\). However, we must extend the notion of state transformation to deal with the possibility that the execution of a command, for certain initial states, may never terminate. For this purpose, we introduce the symbol \(\bot\), usually called "bottom", to denote nontermination, and we take the meanings of commands to be

\[ [-]_{\text{comm}} \in \langle \text{comm} \rangle \rightarrow \Sigma \rightarrow \Sigma_{\bot}, \]

where \(\Sigma_{\bot}\) stands for \(\Sigma \cup \{\bot\}\) (assuming that \(\bot \notin \Sigma\)). Thus the meaning of a command that does not terminate for an initial state \(\sigma\) is a function that maps \(\sigma\) into \(\bot\).

(Many writers use an equivalent formalism where the meaning of a command is a partial function from states to states whose result is undefined for initial states that lead to nontermination. We prefer using \(\Sigma \rightarrow \Sigma_{\bot}\) since it clarifies the generalization to richer languages.)

The effect of an assignment command \(v := e\) is to transform the initial state into a final state that maps \(v\) into the value of \(e\) (in the initial state) and maps all other variables into the same value as in the initial state. Thus we have the semantic equation

\section*{DR SEM EQ: Assignment}

\[ [v := e]_{\text{comm}} \sigma = [\sigma \mid v; [-]_{\text{intexp}} \sigma]. \]

For instance,

\[ [x := x - 1]_{\text{comm}} \sigma = [\sigma \mid x; [-]_{\text{intexp}} \sigma] = [\sigma \mid x; \sigma x - 1] \]
\[ [y := y + x]_{\text{comm}} \sigma = [\sigma \mid y; [-]_{\text{intexp}} \sigma] = [\sigma \mid y; \sigma y + \sigma x]. \]
Notice that the meaning of an assignment command never maps a state into $\bot$, since, in this purely imperative language, expressions always terminate (without error stops).

An even more obvious semantic equation is

\[
\text{DR SEM EQ: skip}
\]

\[
[\text{skip}]_{\text{comm}} \sigma = \sigma.
\]

On the other hand, the sequential composition of commands is complicated by the possibility of nontermination. Naively, one would expect to have the equation

\[
[c_0 ; c_1]_{\text{comm}} \sigma = [c_1]_{\text{comm}}([c_0]_{\text{comm}} \sigma),
\]

but the result of $[c_0]_{\text{comm}} \sigma$ can be $\bot$, which is not in the domain of $[c_1]_{\text{comm}}$; less formally, the equation fails to capture the fact that, if $c_0$ never terminates, then $c_0 ; c_1$ never terminates, regardless of $c_1$.

To solve this problem, we introduce the idea of extending a function to a domain including $\bot$ by mapping $\bot$ into $\bot$. If $f$ is a function from $\Sigma$ to $\Sigma_\bot$, we write $f_{\bot \bot}$ for the function from $\Sigma_{\bot}$ to $\Sigma_\bot$ such that

\[
f_{\bot \bot} \sigma = \text{if } \sigma = \bot \text{ then } \bot \text{ else } f \sigma.
\]

(Here we use a conditional construction in the metalanguage to define the function "by cases". Less formally, $f_{\bot \bot}$ maps $\bot$ to $\bot$ and agrees with $f$ on all other arguments.) Then we have the semantic equation

\[
\text{DR SEM EQ: Sequential Composition}
\]

\[
[c_0 ; c_1]_{\text{comm}} \sigma = ([c_1]_{\text{comm}})_{\bot \bot} ([c_0]_{\text{comm}} \sigma).
\]

For instance,

\[
\begin{align*}
[x := x - 1 ; y := y + x]_{\text{comm}} \sigma \\
&= ([y := y + x]_{\text{comm}})_{\bot \bot} ([x := x - 1]_{\text{comm}} \sigma) \\
&= [y := y + x]_{\text{comm}}[\sigma | x: \sigma x - 1] \\
&= [\sigma | x: \sigma x - 1, y: \sigma y + \sigma x - 1].
\end{align*}
\]

(2.1)

For conditional commands, we have

\[
\text{DR SEM EQ: Conditional}
\]

\[
[\text{if } b \text{ then } c_0 \text{ else } c_1]_{\text{comm}} \sigma = \text{if } [b]_{\text{boolexp}} \sigma \text{ then } [c_0]_{\text{comm}} \sigma \text{ else } [c_1]_{\text{comm}} \sigma.
\]

For instance,

\[
\begin{align*}
[\text{if } x \neq 0 \text{ then } x := x - 1 \text{ else } y := y + x]_{\text{comm}} \sigma \\
&= \text{if } [x \neq 0]_{\text{boolexp}} \sigma \text{ then } [x := x - 1]_{\text{comm}} \sigma \text{ else } [y := y + x]_{\text{comm}} \sigma \\
&= \text{if } \sigma x \neq 0 \text{ then } [\sigma | x: \sigma x - 1] \text{ else } [\sigma | y: \sigma y + \sigma x].
\end{align*}
\]
Notice how this semantic equation captures the fact that the conditional command executes only one of its subcommands: When \([b]_{\text{boolExp}} \sigma\) is true, the equation gives \([c_0]_{\text{comm}} \sigma\) even if \(c_1\) fails to terminate for \(\sigma\) (and vice versa when \([b]_{\text{boolExp}} \sigma\) is false).

So far, our semantic equations have been a straightforward formalization of the idea of state transformation. With \texttt{while} commands, however, we encounter a serious problem whose solution was central to the development of denotational semantics. If one thinks about “unwinding” a \texttt{while} command, it is obvious that

\[
\texttt{while } b \texttt{ do } c \quad \text{and} \quad \texttt{if } b \texttt{ then } (c ; \texttt{ while } b \texttt{ do } c) \texttt{ else } \texttt{skip}
\]

have the same meaning. In other words,

\[
[\texttt{while } b \texttt{ do } c]_{\text{comm}} \sigma = [\texttt{if } b \texttt{ then } (c ; \texttt{ while } b \texttt{ do } c) \texttt{ else } \texttt{skip}]_{\text{comm}} \sigma.
\]

By applying the semantic equations for conditionals, sequential composition, and \texttt{skip} to the right side of this equation, we get

\[
[\texttt{while } b \texttt{ do } c]_{\text{comm}} \sigma = \text{if } [b]_{\text{boolExp}} \sigma \texttt{ then } ([\texttt{while } b \texttt{ do } c]_{\text{comm}} \sigma) \cup ([c]_{\text{comm}} \sigma) \texttt{ else } \sigma.
\]

At first sight, this “unwinding equation” seems to be a plausible semantic equation for the \texttt{while} command. However, although it is an equation that should be satisfied by the meaning of the \texttt{while} command, it is not a semantic equation, since it is not syntax-directed: Because of the presence of \texttt{while } b \texttt{ do } c on the right, it does not describe the meaning of the \texttt{while} command purely in terms of the meanings of its subphrases \(b\) and \(c\). As a consequence, we have no guarantee that the meaning is uniquely determined by this equation.

In general, an equation may have zero, one, or several solutions. In this case, we will find that there is always a solution but, surprisingly, it is not always unique. For example, suppose \(b\) is \(x \neq 0\) and \(c\) is \(x := x - 2\). Then \([x \neq 0]_{\text{boolExp}} \sigma = (\sigma x \neq 0)\) and \([x := x - 2]_{\text{comm}} \sigma = [\sigma | x : \sigma x - 2]\), so that Equation (2.2) reduces to

\[
[\texttt{while } x \neq 0 \texttt{ do } x := x - 2]_{\text{comm}} \sigma = \text{if } \sigma x \neq 0 \texttt{ then } ([\texttt{while } x \neq 0 \texttt{ do } x := x - 2]_{\text{comm}} \sigma) \cup [\sigma | x : \sigma x - 2] \texttt{ else } \sigma.
\]

As the reader may verify, this equation is satisfied by

\[
[\texttt{while } x \neq 0 \texttt{ do } x := x - 2]_{\text{comm}} \sigma = \begin{cases} [\sigma | x : 0] & \text{if } \text{even}(\sigma x) \text{ and } \sigma x \geq 0 \\ \sigma' & \text{if } \text{even}(\sigma x) \text{ and } \sigma x < 0 \\ \sigma'' & \text{if } \text{odd}(\sigma x), \end{cases}
\]

where \(\sigma'\) and \(\sigma''\) can be arbitrary states or \(\perp\). Since \texttt{while } x \neq 0 \texttt{ do } x := x - 2\) does not terminate when \(x\) is either negative or odd, its actual meaning is given by taking \(\sigma' = \sigma'' = \perp\), but there is nothing in the unwinding equation that singles out this solution.
As a more extreme example, suppose \( b \) is \texttt{true} and \( c \) is \texttt{skip}. Then, since \([\texttt{true}]_{\text{assert}} \sigma = \texttt{true}\) and \([\texttt{skip}]_{\text{comm}} \sigma \) is \( \sigma \), the unwinding equation reduces to

\[
[\texttt{while true do skip}]_{\text{comm}} \sigma = [\texttt{while true do skip}]_{\text{comm}} \sigma,
\]

which is satisfied by every function in \( \Sigma \rightarrow \Sigma_\bot \). In this case, we know that \texttt{while true do skip} is a command that never terminates, so that its meaning is the function that maps every state into \( \bot \), but again, this is not a consequence of the unwinding equation.

To overcome this problem, we must introduce the rudiments of domain theory. As we will see in the next two sections, the problem can be solved by making the set of meanings into a domain, which is a certain kind of partial ordering, such that the actual meaning of the \texttt{while} command is the least solution of the unwinding equation.

### 2.3 Domains and Continuous Functions

A domain is a special kind of partially ordered set (see Section A.6 of the Appendix) where the partial order, written \( \sqsubseteq \), is a relationship of “approximation”. When \( x \sqsubseteq y \), we say that \( x \) approximates \( y \), or that \( y \) extends \( x \). The idea (which is quite different from approximation in numerical analysis) is that \( y \) provides at least as much information as \( x \).

A \textit{chain} is a countably infinite increasing sequence, \( x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots \). (Strictly speaking, this is a countable chain, but we will not emphasize this qualification, since we will not consider any other kind of chain.) The least upper bound of a chain is called its \textit{limit}. A chain is said to be \textit{interesting} if it does not contain its own limit or, equivalently, if it contains an infinite number of distinct elements.

A partially ordered set \( P \) is called a \textit{predomain} if every chain of elements of \( P \) has a limit in \( P \). (Note that this requirement is only significant for interesting chains.) A predomain with a least element, which we will denote by \( \bot \), is called a \textit{domain}.

When more than one predomain or domain is involved, we will often decorate symbols such as \( \sqsubseteq \) and \( \bot \) to avoid confusion. For example, the orderings of \( P \) and \( P' \) might be written as \( \sqsubseteq \) and \( \sqsubseteq' \), or even as \( \sqsubseteq_P \) and \( \sqsubseteq_{P'} \).

It should be mentioned that the term “domain” has no universally accepted definition; it has been used with different meanings by different authors, and even by the same author in different publications. Commonly, one requires the existence of limits, not only of chains, but of a more general kind of subset called a “directed set”. Often, additional properties such as “algebraicity” and “bounded completeness” are also imposed.

In general, all such definitions impose stronger requirements than are imposed by the definition we have given above, which will be used throughout this book.
These requirements are actually met by the domains we will be using, but they have no consequences that are significant for the semantic issues we will consider.

Unfortunately, following standard mathematical usage, we will also use the word "domain" with a completely different meaning: to refer to the set over which a function or relation is defined. The key to avoiding confusion is the preposition "of": When we say the domain of something we will be referring to the standard mathematical concept of the domain of a function or relation, but when we speak of a domain per se we will be referring to the above definition. (Thus, oddly, the domain of a function need not be a domain.)

We will often speak of a set "viewed as a predomain". By this we mean that the set is implicitly equipped with the discrete ordering where \( x \leq y \) if and only if \( x = y \). With this ordering, the set is a predomain because its chains cannot contain distinct elements, and so are never interesting.

When \( P \) is a predomain, \( P\perp \) is formed from \( P \) by adding a least element \( \perp \) distinct from any element of \( P \). Except for the extremely uninteresting chain whose every element is \( \perp \), the chains of \( P\perp \) are obtained from the chains of \( P \) by prefixing zero or more occurrences of \( \perp \), and have the same limits. Thus \( P\perp \) is a domain. The operation \((-)\perp \) is often called *lifting*.

Note that, if the set \( \Sigma \) of states is viewed as a predomain, the set \( \Sigma\perp \) that we defined in the previous section becomes an instance of the lifting construction, and thereby acquires a partial ordering:

\[
\sigma_1 \overset{}{\searrow} \sigma_2 \overset{}{\searrow} \sigma_3 \cdots \overset{}{\searrow} \perp
\]

Even in this simple case, the ordering \( \sqsubseteq \) is one of increasing information, in the sense that a computation that terminates in some state provides more information than one that never terminates.

More specifically, \( \Sigma\perp \) is an example of a flat domain, which is a domain \( D \) such that \( D - \{\perp\} \) is discretely ordered. Flat domains are suitable for describing the outcomes of computations that either run on forever without producing output or produce output and immediately terminate. They never contain interesting chains.

Richer domains are needed to describe the outcomes of computations that can repeatedly produce output, perhaps ad infinitum. For a class of computations that produce sequences of integers but never terminate, the appropriate domain is the set of finite and infinite sequences of integers, ordered so that \( s \sqsubseteq s' \) when \( s \) is an initial subsequence of \( s' \). An interesting chain is a sequence of sequences whose lengths grow without limit; each element of such a chain is an initial subsequence of the next, and the limit of the chain is the unique infinite sequence of which
every element is an initial subsequence. For example,

\[ \langle \rangle \subseteq \langle 3, 1 \rangle \subseteq \langle 3, 1, 4 \rangle \subseteq \langle 3, 1, 4, 1, 5 \rangle \subseteq \langle 3, 1, 4, 1, 5, 9 \rangle \subseteq \cdots \]

is an interesting chain whose limit is the sequence of digits of \( \pi \). One can think of such a chain as a sequence of snapshots of the output at increasing times, and the limit as the ultimate total result. (We shall investigate a variation of this kind of domain in Sections 5.2 and 5.3.)

For many computations that produce sequences of integers but never terminate, the user is interested only in the set of integers that appear, but not in the order or number of repetitions of these integers. In this situation, the appropriate domain is the powerset of the integers (the set of sets of integers), with \( s \subseteq s' \) when \( s \subseteq s' \). A chain (interesting or otherwise) is an increasing sequence of sets of integers, and its limit is its union. When the chain elements are all finite sets, one can think of the chain as a sequence of snapshots of the user’s interpretation of the output.

In a more extreme case, the user might only be interested in the number of integers that are output. Then the appropriate domain, sometimes called the vertical domain of the natural numbers, is the set of natural numbers plus the symbol \( \infty \), ordered by their usual total ordering, \( n \subseteq n' \) when \( n \leq n' \):

\[
\begin{array}{c}
\infty \\
\vdots \\
2 \\
1 \\
\bot = 0
\end{array}
\]

Here the interesting chains are those whose elements increase without limit, and their limits are all \( \infty \). (This is the simplest example of a domain that contains interesting chains.)

A function \( f \) from a predomain \( P \) to a predomain \( P' \) is said to be continuous from \( P \) to \( P' \) if it preserves the limits of chains, that is, if, for every chain \( x_0 \subseteq x_1 \subseteq \cdots \) of elements of \( P \), the function \( f \) maps the limit of the chain into an element of \( P' \) that is the least upper bound of \( \{fx_0, fx_1, \ldots \} \).

A continuous function is monotone. To see this, suppose \( f \) is continuous and \( x \subseteq y \), and consider the chain \( x \subseteq y \subseteq y \subseteq \cdots \), whose limit is \( y \). Continuity implies that \( fy \) is the least upper bound of \( \{fx, fy\} \) and, since a least upper bound is an upper bound, \( fx \subseteq fy \).

On the other hand, a monotone function \( f \) will map a chain \( x_0 \subseteq x_1 \subseteq \cdots \) into another chain \( fx_0 \subseteq fx_1 \subseteq \cdots \), which must possess some limit. In this situation, the condition defining continuity can be written as an equality between two domain elements that are known to exist: If \( f \) is a monotone function from a
predomain $P$ to a predomain $P'$, then $f$ is continuous if and only if, for all chains $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ in $P$,

$$f\left(\bigsqcup_{i=0}^{\infty} x_i\right) = \bigsqcup_{i=0}^{\infty} f(x_i). \quad (2.3)$$

Nevertheless, there are monotone functions that are not continuous. For example, suppose $P$ is the vertical domain of the natural numbers, and $P'$ is the two-element domain $\{\bot', \top'\}$. Then the monotone function $fx = \text{if } x = \infty \text{ then } \top' \text{ else } \bot'$,

\[\cdots \quad \top' \quad \vdots \quad 2 \quad 1 \quad \bot' \quad \bot = 0\]

is not continuous, since the limit of $\{0, 1, \ldots\}$ is $\infty$ and $f\infty$ is $\top'$, but the limit of $\{f0, f1, \ldots\} = \{\bot\}$ is $\bot'$.

The extra constraint that a monotone function must satisfy in order to be continuous can be characterized as follows:

**Proposition 2.1** Suppose $f$ is a monotone function from a predomain $P$ to a predomain $P'$. Then $f$ will be continuous if and only if, for all interesting chains $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ in $P$,

$$f\left(\bigsqcup_{i=0}^{\infty} x_i\right) \sqsubseteq \bigsqcup_{i=0}^{\infty} f(x_i).$$

**Proof** We only prove the "if" half of the proposition, since the converse is an immediate consequence of the definition of continuity. Since $\bigsqcup_{i=0}^{\infty} x_i$ is an upper bound of $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ and $f$ is monotone, $f(\bigsqcup_{i=0}^{\infty} x_i)$ is an upper bound of $fx_0 \sqsubseteq' fx_1 \sqsubseteq' \cdots$, and must therefore extend the least upper bound:

$$\bigsqcup_{i=0}^{\infty} f(x_i) \sqsubseteq f(\bigsqcup_{i=0}^{\infty} x_i).$$

Thus the opposite inequality is sufficient to imply continuity. Moreover, the opposite inequality holds trivially when the chain $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ is uninteresting, for then the limit $\bigsqcup_{i=0}^{\infty} x_i$ of the chain belongs to the chain, and it follows that $f(\bigsqcup_{i=0}^{\infty} x_i)$ belongs to $fx_0 \sqsubseteq' fx_1 \sqsubseteq' \cdots$, and thus is bounded by $\bigsqcup_{i=0}^{\infty} fx_i$.

**End of Proof**

When $P$ is finite, or is discretely ordered, or is a flat domain, it contains no interesting chains, so that every monotone function from $P$ to $P'$ is continuous.
Moreover, when $P$ is discretely ordered, all functions from $P$ to $P'$ are monotone, and therefore continuous.

When $P$ and $P'$ are predomains, we write $P \to P'$ for the partially ordered set $P''$ whose elements are the continuous functions from $P$ to $P'$ and whose ordering is the pointwise extension of the ordering on $P'$:

$$f \sqsubseteq'' g \text{ if and only if, for all } x \in P, \ f x \sqsubseteq' g x.$$ 

Under this ordering $P''$ is a predomain and, if $P'$ is a domain, $P''$ is a domain:

**Proposition 2.2** If $P$ and $P'$ are predomains, $P \to P'$ is a predomain in which the limit of a chain $f_0 \sqsubseteq'' f_1 \sqsubseteq'' \cdots$ of functions is the function such that

$$(\bigsqcup_i'' f_i) x = \bigsqcup_i' f_i x$$

for all $x \in P$. Moreover, if $P'$ is a domain, $P \to P'$ is a domain whose least element is the function such that $\bot'' x = \bot'$ for all $x \in P$.

**Proof** Let $f$ be the function such that

$$f x = \bigsqcup_i' f_i x$$

for all $x \in P$. (The limit here must exist since the pointwise ordering implies that the $f_i x$ form a chain.) It is easy to see that, under the pointwise ordering, each $f_i$ approximates $f$ and, for any $g$ that is approximated by all of the $f_i$, $f$ approximates $g$. Thus $f$ will be the least upper bound of the $f_i$ in $P \to P'$ providing it actually belongs to $P \to P'$, that is, if it is a continuous function.

To see this, let $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ be a chain in $P$. Then the definition of $f$ and the continuity of the $f_i$ give

$$f(\bigsqcup_{j=0}^{\infty} x_j) = \bigsqcup_{i=0}^{\infty} f_i(\bigsqcup_{j=0}^{\infty} x_j) = \bigsqcup_{i=0}^{\infty} \bigsqcup_{j=0}^{\infty} f_i x_j,$$

while the definition of $f$ also gives

$$\bigsqcup_{j=0}^{\infty} f x_j = \bigsqcup_{i=0}^{\infty} \bigsqcup_{j=0}^{\infty} f_i x_j.$$ 

By the property of least upper bounds of least upper bounds described in Section A.6 of the Appendix, the right sides of the above equations are both equal to

$$\bigsqcup'\{ f_i x_j \mid i \geq 0 \text{ and } j \geq 0 \},$$

and therefore to each other.

When $P'$ is a domain, the pointwise ordering insures that the constant function yielding $\bot'$ is the least element in $P \to P'$.

END OF PROOF
When $P$ is discrete, $P \to P'$ contains all of the functions from $P$ to $P'$; when $P'$ is discrete, $P \to P'$ is discrete. Thus, as long as discrete ordering is the “default” for viewing sets as predomains, our use of $P \to P'$ to denote the predomain of continuous functions from $P$ to $P'$ does not contradict our use of $S \to S'$ to denote the set of all functions from $S$ to $S'$.

The following proposition establishes that constant and identity functions are continuous, and that composition preserves continuity and is itself a continuous function in each of its arguments:

**Proposition 2.3** Suppose $P$, $P'$, $P''$, $Q$, and $Q'$ are predomains. Then:

(a) A constant function from $P$ to $P'$ is a continuous function from $P$ to $P'$.
(b) The identity function on $P$ is a continuous function from $P$ to $P$.
(c) If $f$ is a continuous function from $P$ to $P'$ and $g$ is a continuous function from $P'$ to $P''$, then $g \cdot f$ is a continuous function from $P$ to $P''$.
(d) If $f$ is a continuous function from $P$ to $P'$, then $- \cdot f$ is a continuous function from $P' \to P''$ to $P \to P''$.
(e) If $g$ is a continuous function from $P'$ to $P''$, then $g \cdot -$ is a continuous function from $P \to P'$ to $P \to P''$.

Imposing a pointwise ordering makes a domain of the set $\Sigma \to \Sigma_\bot$ of meanings of commands that was introduced in the previous section (since $\Sigma_\bot$ is a domain). In particular, the pointwise ordering is

$$f \triangleleft_{\Sigma \to \Sigma_\bot} g \text{ if and only if, for all } \sigma \in \Sigma, f\sigma = \bot \text{ or } f\sigma = g\sigma.$$ 

Again, the ordering is one of increasing information: The computation with meaning $g$ must give the same results as the computation with meaning $f$, except that it may terminate for more initial states. In the next section, we will find that this ordering is the key to understanding the **while** command.

A function from a domain to a domain is said to be **strict** if it preserves least elements, that is, if it maps $\bot$ into $\bot'$. When $f \in P \to P'$, we write $f_\bot$ (called the **lifting** of $f$) for the strict function in $P_\bot \to P'_\bot$ such that

$$f_\bot x = \text{if } x = \bot \text{ then } \bot' \text{ else } fx;$$

when $f \in P \to D$ for a domain $D$, we write $f_\bot$ (sometimes called the **source-lifting** of $f$) for the strict function in $P_\bot \to D$ satisfying the same equation. (Notice that this generalizes the definition in the previous section.) Both of these functions are strict and continuous. On the other hand, the identity injection from $P$ to $P_\bot$, which we denote by $\iota_\bot$, is continuous but not strict. Moreover:
Proposition 2.4 Suppose $P$, $P'$, and $P''$ are predomains, $D$ and $D'$ are domains, $e \in P \rightarrow P'$, $f \in P' \rightarrow P''$, $g \in P'' \rightarrow D$, and $h \in D \rightarrow D'$.

\[ P \xrightarrow{e} P' \xrightarrow{f} P'' \xrightarrow{g} D \xrightarrow{h} D'. \]

Then:

(a) $e_\bot$ is the unique strict extension of $e$ to $P_\bot \rightarrow P'_\bot$.

(b) $g_\bot$ is the unique strict extension of $g$ to $P''_\bot \rightarrow D$.

(c) $(f \cdot e)_\bot = f_\bot \cdot e_\bot$.

(d) $(g \cdot f)_\bot = g_\bot \cdot f_\bot$.

(e) $(h \cdot g)_\bot = h \cdot g_\bot$ when $h$ is strict.

Finally, we note that the set of all subsets of a given set can always be regarded as a domain. When $S$ is a set, we write $\mathcal{P} S$, called the powerset domain, for the domain whose elements are the subsets of $S$, with inclusion $\subseteq$ as the ordering $\sqsubseteq$. In this case, unions are least upper bounds, so that $\bigcup S = \bigcup S$ for any $S \subseteq \mathcal{P} S$ (not just chains). The empty set is the least element.

A function $f$ from $\mathcal{P} S$ to a predomain $P$ is said to be finitely generated if, for all $s \in \mathcal{P} S$, $f(s)$ is the least upper bound of $\{f(s') \mid s' \subseteq s\}$. It can be shown that any finitely generated function is continuous (and, when $S$ is countable, vice versa).

2.4 The Least Fixed-Point Theorem

The following proposition states a fundamental property of domains and continuous functions that underlies the denotational semantics of the while command (and also, as we will see in Sections 11.6 and 14.4, of recursion).

Proposition 2.5 (Least Fixed-Point Theorem) If $D$ is a domain and $f$ is a continuous function from $D$ to $D$, then

\[ x = \bigsqcup_{n=0}^{\infty} f^n \bot \]

is the least fixed-point of $f$. In other words, $fx = x$ and, whenever $fy = y$, $x \subseteq y$.

Proof Clearly $\bot \subseteq f \bot$ and, if $f^n \bot \subseteq f^{n+1} \bot$, then $f^{n+1} \bot = f(f^n \bot) \subseteq f(f^{n+1} \bot) = f^{n+2} \bot$, since $f$ is monotone. Thus, by induction on $n$, $\bot \subseteq f \bot \subseteq f^2 \bot \subseteq \cdots$ is a chain, so that $x$ is well defined.

Then, since $f$ is continuous and $x$ is the least upper bound of a chain,

\[ fx = f(\bigsqcup_{n=0}^{\infty} f^n \bot) = \bigsqcup_{n=0}^{\infty} f^{n+1} \bot = \bigsqcup_{n=1}^{\infty} f^n \bot = \bigsqcup_{n=0}^{\infty} f^n \bot = x, \]
where the penultimate step is justified since one can always prefix \( \bot \) to a chain without changing its least upper bound.

Finally, suppose \( fy = y \). Then \( \bot \subseteq y \) and, if \( f^n \bot \subseteq y \), then \( f^{n+1} \bot = f(f^n \bot) \subseteq fy = y \). Thus, by induction on \( n \), \( y \) is an upper bound on all of the \( f^n \bot \), so that \( x = \bigsqcup_{n=0}^{\infty} f^n \bot \subseteq y \).

Notice that, in the last paragraph of the above proof, a sufficient hypothesis is \( fy \subseteq y \). Thus \( x \) is the least solution of \( fx \subseteq x \) (and is sometimes called the least pre-fixed-point of \( f \)) as well as of \( fx = x \).

We will write \( Y_D \) for the function from \( D \rightarrow D \) to \( D \) such that

\[
Y_D f = \bigsqcup_{n=0}^{\infty} f^n \bot.
\]

Then a more abstract statement of the above proposition is that \( Y_D \) maps continuous functions from \( D \) to \( D \) into their least fixed-points. It can also be shown that \( Y_D \) itself is a continuous function.

We can now use the mathematical machinery we have developed to explain the \texttt{while} command. We know that the meaning of \texttt{while} \( b \) \texttt{do} \( c \) should satisfy the “unwinding” equation (2.2) in Section 2.2:

\[
[\text{while} \ b \ \texttt{do} \ c]_{\text{comm}} \sigma = \text{if} \ [b]_{\text{boolexp}} \sigma \ \text{then} \ ([\text{while} \ b \ \texttt{do} \ c]_{\text{comm}})_{\bot} \ ([c]_{\text{comm}} \sigma) \ \text{else} \ \sigma.
\]

But this equation is simply an assertion that \texttt{while} \( b \) \texttt{do} \( c \) is a fixed point of the function \( F \in (\Sigma \rightarrow \Sigma_{\bot}) \rightarrow (\Sigma \rightarrow \Sigma_{\bot}) \) such that

\[
F f \sigma = \text{if} \ [b]_{\text{boolexp}} \sigma \ \text{then} \ f_{\bot} ([c]_{\text{comm}} \sigma) \ \text{else} \ \sigma.
\]

It can be shown that this function is continuous. Thus, the least fixed-point theorem assures us that the unwinding equation has a solution, and it gives us a criteria — leastness — for picking out a particular solution. If we make the leap of faith that the least solution is the right solution, then we have

\[
\text{DR SEM EQ: while}
\]

\[
[\text{while} \ b \ \texttt{do} \ c]_{\text{comm}} = Y_{\Sigma \rightarrow \Sigma_{\bot}} F
\]

where \( F f \sigma = \text{if} \ [b]_{\text{boolexp}} \sigma \ \text{then} \ f_{\bot} ([c]_{\text{comm}} \sigma) \ \text{else} \ \sigma \).

We cannot prove our leap of faith that this semantic equation is correct, because we have no other rigorous definition of the \texttt{while} command. But we can give an informal argument that appeals to the reader’s intuitive operational understanding. Let \( w_0, w_1, \ldots \) be the sequence of commands

\[
w_0 \overset{\text{def}}{=} \text{while true do skip}
\]

\[
w_i + 1 \overset{\text{def}}{=} \text{if } b \ \texttt{then} \ (c ; w_n) \ \texttt{else} \ \texttt{skip}.
\]
The command $w_0$ obviously never terminates, so that $[w_0]_{\text{comm}} = \bot$, and it is easy to work out that $[w_{i+1}]_{\text{comm}} = F[w_i]_{\text{comm}}$. Thus
\[
[w_i]_{\text{comm}} = F^i \bot.
\]

Now comes the crucial informal argument: Starting in a particular state $\sigma$, the two commands $w_i$ and $\textbf{while } b \textbf{ do } c$ will behave the same way, unless the $\textbf{while}$ command executes the test $b$ at least $i$ times, in which case $w_i$ will not terminate. Thus:

- If $\textbf{while } b \textbf{ do } c$ terminates after testing $b$ exactly $n$ times, then
  \[
  [w_i]_{\text{comm}} \sigma = \begin{cases} 
  \bot & \text{when } i < n \\
  [\textbf{while } b \textbf{ do } c]_{\text{comm}} \sigma & \text{when } i \geq n.
  \end{cases}
  \]

- If $\textbf{while } b \textbf{ do } c$ does not terminate, then, for all $i$,
  \[
  [w_i]_{\text{comm}} \sigma = \bot = [\textbf{while } b \textbf{ do } c]_{\text{comm}} \sigma.
  \]

In either case, however, $[\textbf{while } b \textbf{ do } c]_{\text{comm}} \sigma$ is the limit of the chain $[w_0]_{\text{comm}} \sigma$, $[w_1]_{\text{comm}} \sigma$, \ldots,
\[
[w_i]_{\text{comm}} \sigma = \bigcup_{n=0}^{\infty} [w_n]_{\text{comm}} \sigma
\]
and, by Proposition 2.2 in the previous section and the least fixed-point theorem,
\[
[w_i]_{\text{comm}} = \bigcup_{n=0}^{\infty} [w_n]_{\text{comm}} = \bigcup_{n=0}^{\infty} F^n \bot = Y_{\Sigma \to \Sigma \bot} F.
\]

As a trivial example, when $b$ is $\textbf{true}$ and $c$ is $\textbf{skip}$ the function $F$ is the identity function on $\Sigma \to \Sigma \bot$, whose least fixed-point is the function mapping every state into $\bot$, which is indeed the meaning of $\textbf{while } \textbf{true } \textbf{do } \textbf{skip}$.

As a nontrivial example, using Equation (2.1) in Section 2.2 we find
\[
[w_x \neq 0 \textbf{ do } (x := x - 1 ; y := y + x)]_{\text{comm}} = Y_{\Sigma \to \Sigma \bot} F,
\]
where
\[
F f \sigma = \textbf{if } \sigma x \neq 0 \textbf{ then } f_{\bot} [\sigma \mid x: \sigma x - 1 \mid y: \sigma y + \sigma x - 1] \textbf{ else } \sigma.
\]

There is no universal method for converting such an equation into a more explicit form. In this case, however, our informal expectation is that the "approximate" $\textbf{while}$ command $w_n$, whose meaning is $F^n \bot$, will terminate when $0 \leq x < n$, and when it terminates it will set $x$ to zero and increase $y$ by each of the integers from $x - 1$ down to zero. It is known that the sum of these integers is $(x \times (x - 1) \div 2)$. Thus we expect
\[
F^n \bot \sigma = \textbf{if } 0 \leq \sigma x < n \textbf{ then } [\sigma \mid x: 0 \mid y: \sigma y + \sigma x \times (\sigma x - 1) \div 2] \textbf{ else } \bot.
\]
In fact, we can prove this equation by induction on \( n \). When \( n = 0 \), the equation follows from \( \bot \sigma = \bot \). The induction step is

\[
F^{n+1} \bot \sigma = F(F^n \bot) \sigma
= \text{if } \sigma x \neq 0 \text{ then } (F^n(\bot))_{\bot} [x: \sigma x - 1 \mid y: \sigma y + \sigma x - 1] \text{ else } \sigma
= \text{if } \sigma x \neq 0 \text{ then }
\quad \text{if } 0 \leq \sigma x - 1 < n \text{ then }
\qquad [x: 0 \mid y: \sigma y + \sigma x - 1 + (\sigma x - 1) \times (\sigma x - 2) \div 2]
\quad \text{else } \bot
\text{else } \sigma
= \text{if } 0 \leq \sigma x < n + 1 \text{ then } [x: 0 \mid y: \sigma y + \sigma x \times (\sigma x - 1) \div 2] \text{ else } \bot.
\]

Now consider the chain \( F^0 \bot \sigma, F^1 \bot \sigma, \ldots \). If \( \sigma x \geq 0 \), the chain will consist of some finite number of occurrences of \( \bot \), followed by an infinite number of occurrences of \( [x: 0 \mid y: \sigma y + \sigma x \times (\sigma x - 1) \div 2] \), which will be the limit of the chain. On the other hand, if \( \sigma x < 0 \), the chain will consist entirely of \( \bot \), which will be its limit. Thus

\[
[\text{while } x \neq 0 \text{ do } (x := x - 1; y := y + x)]_{\text{comm}} \sigma
= \mathcal{Y}_{\Sigma \rightarrow \Sigma} F \sigma
= \bigsqcup_{n=0}^{\infty} F^n \bot \sigma
= \text{if } \sigma x \geq 0 \text{ then } [x: 0 \mid y: \sigma y + \sigma x \times (\sigma x - 1) \div 2] \text{ else } \bot.
\]

A very different application of the least fixed-point theorem is to abstract syntax definitions. In Section 1.1, we saw that such a definition can be regarded as a system of equations of the form

\[
\begin{align*}
{s_0}^{(0)} &= \{\} & \cdots & {s_{n-1}}^{(0)} = \{\}
{s_0}^{(j+1)} = f_0(s_0^{(j)}, \ldots, s_{n-1}^{(j)}) & \cdots & {s_{n-1}}^{(j+1)} = f_{n-1}(s_0^{(j)}, \ldots, s_{n-1}^{(j)})
{s_0} = \bigcup_{j=0}^{\infty} s_0^{(j)} & \cdots & {s_{n-1}} = \bigcup_{j=0}^{\infty} s_{n-1}^{(j)}.
\end{align*}
\]
where, for $0 \leq i \leq n - 1$ and $j \geq 0$, the $s^{(j)}_i$ and $s_i$ belong to the powerset $\mathcal{P}(\mathcal{P})$ of the universe $\mathcal{P}$ of all phrases and the $f_i$ are functions from $(\mathcal{P}(\mathcal{P}))^n$ to $\mathcal{P}(\mathcal{P})$.

Suppose we define the elements

$$s^{(j)} = \{s^{(j)}_0, \ldots, s^{(j)}_{n-1}\} \quad s = \{s_0, \ldots, s_{n-1}\}$$

of the domain $(\mathcal{P}(\mathcal{P}))^n$, and the function $f = f_0 \otimes \cdots \otimes f_{n-1}$ from $(\mathcal{P}(\mathcal{P}))^n$ to $(\mathcal{P}(\mathcal{P}))^n$. Then the above equations can be rewritten more succinctly as

$$s^{(0)} = \{\}, \ldots, \{\} = \bot \quad s^{(j+1)} = f(s^{(j)}) \quad s = \bigsqcup_{j=0}^{\infty} s^{(j)},$$

or just

$$s = \bigsqcup_{j=0}^{\infty} f^j \bot.$$

By the least fixed-point theorem, $s$ is the least solution of $s = fs$. In terms of the original equations, $s = \{s_0, \ldots, s_{n-1}\}$ is the family of least sets satisfying the original equations with the superscripts $(j)$ omitted.

Of course, this depends on the function $f$ being continuous. In fact, the continuity of $f$ stems from the fact that each $f_i$ is finitely generated in each of its $n$ arguments, which in turn stems from the fact that the constructors of the abstract syntax have a finite number of arguments.

### 2.5 Variable Declarations and Substitution

In this section, we extend the simple imperative language by adding *variable declarations*. The abstract syntax is given by the production

$$\langle \text{comm} \rangle ::= \text{newvar} \langle \text{var} \rangle := \langle \text{intexp} \rangle \text{ in } \langle \text{comm} \rangle$$

(We will use the term "declaration" for the subphrase `newvar (var) := (intexp)` and also for the command that begins with this subphrase, which many authors call a "block".) The concrete syntax of this command is similar to the conditional and *while* constructions: It extends to the first stopping symbol or to the end of the enclosing phrase.

Semantically, `newvar v := e in c` initializes the variable $v$ to the value of $e$, executes $c$, and, if $c$ terminates, resets $v$ to whatever value it had before initialization. This behavior is captured by the semantic equation

$$[\text{newvar } v := e \text{ in } c]_{\text{comm},\sigma}$$

$$= \text{if } [c]_{\text{comm}}[\sigma | v : [e]_{\text{intexp},\sigma}] = \bot \text{ then } \bot \text{ else }$$

$$[[c]_{\text{comm}}[\sigma | v : [e]_{\text{intexp},\sigma}] | v : \sigma v]$$

or, more abstractly,
DR SEM EQ: Variable Declaration

\[ [\text{newvar } v := e \text{ in } c]_{\text{comm}} \sigma = (\lambda \sigma' \in \Sigma. [\sigma' \mid v: \sigma' v]) \sqcup (\llbracket c \rrbracket_{\text{comm}}(\sigma \mid v: \llbracket e \rrbracket_{\text{intexp}} \sigma)). \]

(Here we have used a typed abstraction in the metalanguage. As explained in Section A.3 of the Appendix, \( \lambda \sigma' \in \Sigma. [\sigma' \mid v: \sigma v] \) stands for the function \( f \) such that \( f \sigma' = [\sigma' \mid v: \sigma v] \) for all \( \sigma' \in \Sigma \).)

Variable declarations are a simple example of a construct that improves the scalability of programs. Although they are of little value in small programs, they are essential for the readability of large ones. When a program involves hundreds of variables, it is vital that the programmer be able to indicate the region of the program to which each variable is local. (Notice that this notion of locality disconnects the behavior of the variable within the region where it is declared from the behavior of a variable with the same name outside the region. A precise definition of locality will be formalized in Section 3.5.)

We have made a specific design choice in allowing a newly declared variable to be initialized to the value of an arbitrary integer expression. Some languages use a default initialization, typically to zero, but this is unnecessarily restrictive. Many others leave the initialization unspecified, with the pragmatic intention that the initialization may be whatever value happens to lie in the newly allocated word of storage. Such a value, however, will usually be logically unrelated to the variable being declared, and may even be a machine instruction or (in some operating environments) data or code from someone else's computation. This can make debugging extraordinarily difficult, since the behavior of a program that depends on unspecified initial values will be inexplicable in terms of any reasonable semantics, and intelligible only when one understands the details of storage allocation, and perhaps code generation, by the compiler. Indeed, in some environments, the initialization may be determined by factors that make program behavior irreproducible.

In \text{newvar } v := e \text{ in } c, \) the occurrence of \( v \) is a binder whose scope is \( c \) (but not \( e \)). In contrast, in the assignment command \( v := e, \) the occurrence of \( v \) is not a binder. Thus the variables occurring free in a command are given by

\[
\begin{align*}
\text{FV}_{\text{comm}}(v := e) &= \{v\} \cup \text{FV}_{\text{intexp}}(e) \\
\text{FV}_{\text{comm}}(\text{skip}) &= \{\} \\
\text{FV}_{\text{comm}}(c_0 ; c_1) &= \text{FV}_{\text{comm}}(c_0) \cup \text{FV}_{\text{comm}}(c_1) \\
\text{FV}_{\text{comm}}(\text{if } b \text{ then } c_0 \text{ else } c_1) &= \text{FV}_{\text{boolexp}}(b) \cup \text{FV}_{\text{comm}}(c_0) \cup \text{FV}_{\text{comm}}(c_1) \\
\text{FV}_{\text{comm}}(\text{while } b \text{ do } c) &= \text{FV}_{\text{boolexp}}(b) \cup \text{FV}_{\text{comm}}(c) \\
\text{FV}_{\text{comm}}(\text{newvar } v := e \text{ in } c) &= (\text{FV}_{\text{comm}}(c) - \{v\}) \cup \text{FV}_{\text{intexp}}(e),
\end{align*}
\]
where $\text{FV}_{\text{inexp}}$ is the same as in the previous chapter and $\text{FV}_{\text{boolexp}}$ is the same as $\text{FV}_{\text{assert}}$, but restricted to quantifier-free expressions.

One can also define $\text{FA}(c) \subseteq \text{FV}_{\text{comm}}(c)$ to be the set of variables that occur free on the left of an assignment operation in $c$:

$$\text{FA}(v := e) = \{v\}$$

$$\text{FA}(\text{skip}) = \{\}$$

$$\text{FA}(c_0 ; c_1) = \text{FA}(c_0) \cup \text{FA}(c_1)$$

$$\text{FA}(\text{if } b \text{ then } c_0 \text{ else } c_1) = \text{FA}(c_0) \cup \text{FA}(c_1)$$

$$\text{FA}(\text{while } b \text{ do } c) = \text{FA}(c)$$

$$\text{FA}(\text{newvar } v := e \text{ in } c) = \text{FA}(c) - \{v\}.$$

Substitution into integer and boolean expressions is defined as in the previous chapter. (Indeed, in the absence of quantifiers it is trivial.) But substitution into commands is much more constrained, since the substitution of an expression that is not a variable for the occurrence of a variable on the left side of an assignment command would produce a syntactically illegal phrase, such as $(x := x + 1)/x \rightarrow 10$, which is $10 := 10 + 1$ or $(x := x + 1)/x \rightarrow y * z$, which is $y * z := y * z + 1$.

However, if we limit the substitution map $\delta$ to a function that yields variables, so that $\delta \in \langle \text{var} \rangle \rightarrow \langle \text{var} \rangle$, then we can define

$$(v := e)/\delta = (\delta v) := (e/\delta)$$

$$\text{skip}/\delta = \text{skip}$$

$$(c_0 ; c_1)/\delta = (c_0/\delta) ; (c_1/\delta)$$

$$(\text{if } b \text{ then } c_0 \text{ else } c_1)/\delta = \text{if } (b/\delta) \text{ then } (c_0/\delta) \text{ else } (c_1/\delta)$$

$$(\text{while } b \text{ do } c)/\delta = \text{while } (b/\delta) \text{ do } (c/\delta)$$

$$(\text{newvar } v := e \text{ in } c)/\delta = \text{newvar } v_{\text{new}} := (e/\delta) \text{ in } (c/[\delta | v : v_{\text{new}}])$$

where

$$v_{\text{new}} \notin \{\delta w | w \in \text{FV}_{\text{comm}}(c) - \{v\}\}.$$
The properties of free variables and substitution given by Propositions 1.1 to 1.4 in Section 1.4 remain true when $\theta$ is “integer expression” or “boolean expression”, and the syntactic properties given by Proposition 1.2 also hold when $\theta$ is “command”. But the semantic properties of free variables and substitution are more complicated for commands.

For instance, in the absence of nontermination, $[c]_{\text{comm}} \sigma$ is a state that depends on the entire state $\sigma$, not just the part of $\sigma$ that acts on the free variables of $c$. However, suppose we consider $[c]_{\text{comm}} \sigma w$ for a particular variable $w$. If $w$ is a free variable of $c$, then $[c]_{\text{comm}} \sigma w$ depends only on the part of $\sigma$ that acts on the free variables. On the other hand, if $w$ is not a free variable, indeed if $w$ is not assigned by $c$, then $[c]_{\text{comm}} \sigma w$ is the same as $\sigma w$.

For example, consider $[x := x + z ; y := x + y]_{\text{comm}} \sigma$, where $\sigma = [\sigma_0 | x: a | y: b | z: c]$. If $w$ is $x$, $y$, or $z$, then $[x := x + z ; y := x + y]_{\text{comm}} \sigma w$ depends on $a$, $b$, and $c$, but not $\sigma_0$; while if $w$ is not $x$ or $y$, then $[x := x + z ; y := x + y]_{\text{comm}} \sigma w = \sigma w$.

In general, one can show

**Proposition 2.6 (Coincidence Theorem for Commands)**

(a) If $\sigma$ and $\sigma'$ are states such that $\sigma w = \sigma' w$ for all $w \in \text{FV}_{\text{comm}}(c)$, then either $[c]_{\text{comm}} \sigma = [c]_{\text{comm}} \sigma'$ is equal to $\bot$ or else $[c]_{\text{comm}} \sigma$ and $[c]_{\text{comm}} \sigma'$ are states such that $([c]_{\text{comm}} \sigma) w = ([c]_{\text{comm}} \sigma') w$ for all $w \in \text{FV}_{\text{comm}}(c)$.

(b) If $[c]_{\text{comm}} \sigma \neq \bot$, then $([c]_{\text{comm}} \sigma) w = \sigma w$ for all $w \notin \text{FA}(c)$.

Surprisingly — and disturbingly — the substitution theorem fails for commands. In fact, when a substitution carries distinct variables into the same variable (in which case the distinct variables are said to become aliases), it can map commands with the same meaning into commands with different meanings. For example,

$$x := x + 1 ; y := y \times 2 \quad \text{and} \quad y := y \times 2 ; x := x + 1$$

have the same meaning, but the aliasing substitution of $z$ for both $x$ and $y$ maps these commands into

$$z := z + 1 ; z := z \times 2 \quad \text{and} \quad z := z \times 2 ; z := z + 1,$$

which have different meanings.

Aliasing is an inherent subtlety of imperative programming that is a rich source of programming errors. A less trivial example is

$$y := 1 ; \text{while} \ x > 0 \ \text{do} \ (y := y \times x ; x := x - 1).$$

When $x \geq 0$, this command sets $y$ to the factorial of the initial value of $x$. But the command obtained by substituting the same variable $z$ for both $x$ and $y$,

$$z := 1 ; \text{while} \ z > 0 \ \text{do} \ (z := z \times z ; z := z - 1),$$

does not set $z$ to the factorial of the initial value of $z$. (We will see in Chapter 13...
that aliasing problems are exacerbated when an imperative language is extended with a procedure mechanism that uses call by name or call by reference.

Even though the substitution theorem fails in general, one can still give a theorem about substitutions that do not create aliases since they map distinct variables into distinct variables:

**Proposition 2.7 (Substitution Theorem for Commands)** Suppose $c$ is a command, $\delta \in \langle \text{var} \rangle \to \langle \text{var} \rangle$, and $V$ is a set of variables, such that $\text{FV}_{\text{comm}}(c) \subseteq V$ and $\delta w \not= \delta w'$ whenever $w$ and $w'$ are distinct members of $V$. If $\sigma w = \sigma'(\delta w)$ for all $w \in V$, then either $[c/\delta]_{\text{comm}}\sigma' = [c]_{\text{comm}}\sigma = \bot$ or else $[c/\delta]_{\text{comm}}\sigma'$ and $[c]_{\text{comm}}\sigma$ are states such that $([c/\delta]_{\text{comm}}\sigma')(\delta w) = ([c]_{\text{comm}}\sigma)w$ for all $w \in V$.

**Proof** As with the substitution theorem for predicate logic (Proposition 1.3 in Section 1.4), the proof is by structural induction (on $c$) with a case analysis on the constructors of the abstract syntax, and the delicate part of the argument is the case of the binding constructor. We limit our exposition to this case and, to focus on the significant aspects, we only consider initialization to zero.

Suppose $c$ is $\text{newvar} \ v := 0 \ in \ c'$, $\text{FV}_{\text{comm}}(c) \subseteq V$, $\delta w \not= \delta w'$ whenever $w$ and $w'$ are distinct members of $V$, and $\sigma w = \sigma'(\delta w)$ for all $w \in V$. Assume that $v_{\text{new}} \not\in \{\delta w \mid w \in \text{FV}_{\text{comm}}(c') \setminus \{v\}\}$. Consider the equation

$$[\sigma \mid v:0]w = [\sigma' \mid v_{\text{new}}:0](\delta \mid v:v_{\text{new}})w).$$

This equation holds when $w = v$, and it also holds for all $w \in \text{FV}_{\text{comm}}(c') \setminus \{v\}$ since then $v_{\text{new}} \not= \delta w$, and thus $\sigma w = \sigma'(\delta w) = [\sigma' \mid v_{\text{new}}:0](\delta w)$. Thus the equation holds for all $w \in \text{FV}_{\text{comm}}(c')$.

Moreover, if $w$ and $w'$ are distinct variables in $\text{FV}_{\text{comm}}(c')$, then

$$[\delta \mid v:v_{\text{new}}]w \not= [\delta \mid v:v_{\text{new}}]w',$$

since either $w$ and $w'$ are both in $\text{FV}_{\text{comm}}(c') \setminus \{v\} = \text{FV}_{\text{comm}}(c) \subseteq V$, in which case $[\delta \mid v:v_{\text{new}}]w = \delta w \not= \delta w' = [\delta \mid v:v_{\text{new}}]w'$, or one of these variables, say $w'$, is $v$ and the other is in $\text{FV}_{\text{comm}}(c') \setminus \{v\}$, in which case $[\delta \mid v:v_{\text{new}}]w = \delta w \not= v_{\text{new}} = [\delta \mid v:v_{\text{new}}]w'$. Thus, we can apply the induction hypothesis, with $c$ replaced by $c'$, $V$ by $\text{FV}_{\text{comm}}(c')$, $\delta$ by $[\delta \mid v:v_{\text{new}}]$, $\sigma'$ by $[\sigma' \mid v_{\text{new}}:0]$, and $\sigma$ by $[\sigma \mid v:0]$. We find that

$$[c'/[\delta \mid v:v_{\text{new}}]_{\text{comm}}[\sigma' \mid v_{\text{new}}:0] \text{ and } [c']_{\text{comm}}[\sigma \mid v:0]$$

are either both equal to $\bot$ or are states satisfying

$$([c'/[\delta \mid v:v_{\text{new}}]_{\text{comm}}[\sigma' \mid v_{\text{new}}:0])([\delta \mid v:v_{\text{new}}]w) = ([c']_{\text{comm}}[\sigma \mid v:0])w$$

for all $w \in \text{FV}_{\text{comm}}(c')$. 


Now consider the case where termination occurs, and suppose \( w \in \text{FV}_{\text{comm}}(c) = \text{FV}_{\text{comm}}(c') - \{v\} \). Then the definition of substitution, the semantic equation for variable declarations, the facts that \( v_{\text{new}} \neq \delta w \) and \( v \neq w \), the above equation, and again \( v \neq w \) and the semantic equation for variable declarations give

\[
(\sem{c/\delta}_{\text{comm}}^{\sigma'})(\delta w) = (\sem{\text{newvar } v := 0 \text{ in } c'/\delta}_{\text{comm}}^{\sigma'})(\delta w)
= (\sem{\text{newvar } v_{\text{new}} := 0 \text{ in } (c'/\delta \mid v: v_{\text{new}})}_{\text{comm}}^{\sigma'})(\delta w)
= (\sem{c'/\delta \mid v: v_{\text{new}}}_{\text{comm}}^{\sigma'}(\sigma' \mid v_{\text{new}}: 0))(\delta w)
= (\sem{c'/\delta \mid v: v_{\text{new}}}_{\text{comm}}^{\sigma'}(\sigma' \mid v_{\text{new}}: 0))(\delta w)
= (\sem{[c']_{\text{comm}}^{\sigma \mid v: 0}})(\delta w)
= (\sem{[c']_{\text{comm}}^{\sigma \mid v: 0}})(\delta w)
= (\sem{\text{newvar } v := 0 \text{ in } c'}_{\text{comm}}^{\sigma})w
= (\sem{c}_{\text{comm}}^{\sigma})w.
\]

On the other hand, in the case of nontermination, the parenthesized expressions immediately following the equality signs in the above display are all equal to \( \bot \).

To complete the proof, we must extend the argument for the case of termination to variables in \( V \) that do not occur free in \( c \). Suppose \( w \in V - \text{FV}_{\text{comm}}(c) \). If \( \delta w \) occurred free in \( c/\delta \), then, by Proposition 1.2(c) in Section 1.4, there would be a \( w_{\text{free}} \in \text{FV}_{\text{comm}}(c) \) such that \( \delta w = \delta w_{\text{free}} \); but then \( w \) and \( w_{\text{free}} \) would be distinct variables in \( V \), so that \( \delta w \neq \delta w_{\text{free}} \), which would be contradictory. Thus \( \delta w \notin \text{FV}_{\text{comm}}(c/\delta) \). Then, by Proposition 2.6(b),

\[
(\sem{c/\delta}_{\text{comm}}^{\sigma'})(\delta w) = \sigma'(\delta w) = \sigma w = (\sem{c}_{\text{comm}}^{\sigma})w.
\]

**END OF PROOF**

The above proposition and its proof illustrate a phenomenon that is typical of subtle induction arguments. Clearly, two important special cases of the proposition arise when \( V \) is the set of all variables and, at the opposite extreme, when \( V = \text{FV}_{\text{comm}}(c) \). Neither of these cases, however, can be proved directly by structural induction on \( c \). To prove the case where \( V \) is the set of all variables, one would have to infer that the injectiveness of \( \delta \) implies the injectiveness of \( \lfloor \delta \mid v: v_{\text{new}} \rfloor \), which is not true in general. To prove the case where \( V = \text{FV}_{\text{comm}}(c) \), in order to use the induction hypothesis when \( c \) is \( c_0 ; c_1 \) (a case that is omitted in the above proof), one would need \( \text{FV}_{\text{comm}}(c_0) = \text{FV}_{\text{comm}}(c_1) = \text{FV}_{\text{comm}}(c) \), which is also not true in general. Thus, to prove either of the special cases, one must generalize to the above proposition before employing induction.

Fortunately, the phenomenon of aliasing does not affect the fundamental property of binding that renaming preserves meaning:
Proposition 2.8 (Renaming Theorem for Commands) If
\[ v_{\text{new}} \notin \text{FV}_{\text{comm}}(c') - \{v\}, \]
then
\[ \left[ \text{newvar } v_{\text{new}} := e \text{ in } (c'/v \rightarrow v_{\text{new}}) \right]_{\text{comm}} \sigma = \left[ \text{newvar } v := e \text{ in } c' \right]_{\text{comm}} \sigma. \]

PROOF We repeat the argument in the previous proof, for the special case where \( V \) is the set \( \langle \text{var} \rangle \) of all variables, \( \delta \) is the identity function \( I_{\langle \text{var} \rangle} \), and \( \sigma = \sigma' \). (As in the previous proof, for simplicity we only consider the case where \( e = 0 \).) Then either
\[ \left( \left[ \text{newvar } v_{\text{new}} := 0 \text{ in } (c'/v \rightarrow v_{\text{new}}) \right]_{\text{comm}} \sigma \right)^w = \left( \left[ \text{newvar } v := 0 \text{ in } c' \right]_{\text{comm}} \sigma \right)^w \]
for all \( w \in \langle \text{var} \rangle \), or the entities in large parentheses are both equal to \( \perp \). In either case, the entities in large parentheses are equal. The only assumption made about \( v_{\text{new}} \) is that it is distinct from \( \delta w \) for all \( w \in \text{FV}_{\text{comm}}(c') - \{v\} \).

As we gradually extend the imperative language in later chapters, the above proposition and its analogues for other binding constructions will remain true. Of course, one must extend the definition of the functions \( \text{FV} \) and \( \text{FA} \) to encompass the new linguistic constructions. We will give explicit equations for binding constructions and constructions that have variables as immediate subphrases, but we will omit the multitude of equations that merely equate \( \text{FV} \) or \( \text{FA} \) of some construction to the union of the same function applied to the immediate subphrases.

2.6 Syntactic Sugar: The for Command

Many imperative languages provide some form of command that iterates a subcommand over an interval of integers. For example, we might introduce a for command with the syntax
\[ \langle \text{comm} \rangle ::= \text{for } \langle \text{var} \rangle := \langle \text{intexp} \rangle \text{ to } \langle \text{intexp} \rangle \text{ do } \langle \text{comm} \rangle \]
such that for \( v := e_0 \) to \( e_1 \) do \( c \) sets \( v \) to each integer from \( e_0 \) to \( e_1 \) inclusive, in increasing order, and after each such assignment executes the subcommand \( c \).

Rather than giving a semantic equation for the for command, it is simpler to define its meaning by giving an equivalent command in the language we have already defined:
\[ \text{for } v := e_0 \text{ to } e_1 \text{ do } c \overset{\text{def}}{=} (v := e_0 ; \text{while } v \leq e_1 \text{ do } (c ; v := v + 1)). \]

More precisely, this equation defines the meaning of for commands in the following sense: By repeatedly replacing an occurrence of the left side of an instance
of the equation by the right side of the instance, one can translate any phrase of our language containing for commands into one that no longer contains such commands.

When a language construct can be defined in this way, it is often called syntactic sugar, and the translation that eliminates it is called desugaring. This perspicuous terminology was coined by Peter Landin, who used the method to reduce functional languages to the lambda calculus (as we will describe in Section 11.3).

Obviously, constructions that can be defined as syntactic sugar do not enhance the expressiveness of a language, that is, the variety of computational processes that can be described in the language. However, such constructions may allow certain processes to be described more succinctly or intelligibly.

Despite its seeming simplicity, the for command is notorious for the variety of subtly different designs that are possible, as well as the variety of design mistakes that can encourage programming errors. In the above definition, we avoided one common mistake by prescribing that for v := e0 to e1 do c will not execute c at all when there are no integers in the interval from e0 to e1, which will occur when e0 > e1. In contrast, the DO command in early versions of Fortran always executed its body at least once. In many applications, this forced the programmer to introduce a special branch to handle the (often rare but essential) case of iterating over the empty interval.

There are other ways, however, in which the above definition is unsatisfactory. For instance, the execution of for v := e0 to e1 do c will have the "side effect" of resetting the control variable v. But usually the programmer intends v to be a local variable of the for command. Thus a better version of the for command is defined by

\[
\text{for } v := e_0 \text{ to } e_1 \text{ do } c \overset{\text{def}}{=} \text{newvar } v := e_0 \text{ in while } v \leq e_1 \text{ do } (c ; v := v + 1).
\]

Both of these versions, however, have the defect that the upper-limit expression \( e_1 \) is reevaluated after every execution of \((c ; v := v + 1)\). Thus, if \( e_1 \) contains \( v \) or any variable assigned by \( c \), then \( v \) will be tested against a changing upper bound. As an extreme example, for \( x := 1 \text{ to } x \text{ do } \text{skip} \) will never terminate.

It is better to test the control variable against a fixed upper bound that is the value of \( e_1 \) when the for command begins execution, as in the definition

\[
\text{for } v := e_0 \text{ to } e_1 \text{ do } c \overset{\text{def}}{=} \text{newvar } w := e_1 \text{ in newvar } v := e_0 \text{ in while } v \leq w \text{ do } (c ; v := v + 1),
\]

where \( w \) is distinct from \( v \) and does not occur free in either \( e_0 \) or \( c \). This version also has the advantage that \( e_1 \), like \( e_0 \), is excluded from the scope of the binding of \( v \).
A final problem is that, if the subcommand $c$ changes the control variable $v$, then successive executions of $c$ may not occur for consecutive values of $v$. For example, $\text{for } x := 1 \text{ to } 10 \text{ do } (c' ; x := 2 \times x)$ will execute $c'$ for $x = 1, 3, \text{ and } 7$. This grotesqueness can be avoided by imposing the restriction $v \notin FA(c)$ on the $\text{for}$ command.

It is important to understand why our successive redefinitions of the $\text{for}$ command are improvements. The utility of a properly designed $\text{for}$ command is that, in comparison with the $\text{while}$ command, it describes a more restricted form of iteration that is easier to reason about. For instance, if the $\text{for}$ command is defined by the final equation above, with the restriction $v \notin FA(c)$, and if $c$ always terminates, then $\text{for } v := e_0 \text{ to } e_1 \text{ do } c$ will always execute $c$ the number of times that is the size of the interval $e_0$ to $e_1$, where $e_0$ and $e_1$ are evaluated at the beginning of execution of the $\text{for}$ command.

2.7 Arithmetic Errors

The language presented in this chapter raises two possibilities for arithmetic error: division by zero and overflow. In this section, we first consider the language design issues raised by these errors, and then the treatment of these issues in terms of the semantics we have presented.

There is a fundamental difference between these kinds of errors: A program that executes an overflow may run without error on a machine with a larger range of representable integers, but a program that divides by zero would not be correct even on a machine with perfect arithmetic. Nevertheless, the ways of treating these errors (and the relative merits of these ways) are similar in both cases.

Basically, the language designer must choose between checking for these errors and reporting them to the user (either by giving an error message or, more flexibly, by the kind of exception mechanism to be discussed in Exercises 5.3 and 5.4, and in Section 13.7), or ignoring the errors and taking whatever results are produced by the underlying hardware. (One can provide both options within a single language, but this merely shifts an essential decision from the language designer to the programmer.)

There are extreme cases where detecting arithmetic errors is not worth the cost in execution time. On the one hand, there are programs for which occasional erroneous behavior is tolerable, for example, programs whose output will be checked by the user or by another program. At the other extreme are programs, particularly in real-time processing, where an error message is just as intolerable as an incorrect answer, so that there is no safe alternative to proving that errors will not occur.

In the vast majority of computer applications, however, a failure to produce a result is far less serious than producing an erroneous result (especially if an error
message is given, as opposed to silent nontermination). Indeed, reasoning about overflow is so difficult that it is common programming practice to ignore the issue, write programs that would behave correctly on a machine with perfect arithmetic, and rely on error checking to be sure that the machine does not deviate from such perfection. (Notice that we are not considering floating-point arithmetic, where the problem of controlling precision can be far more intractable than that of overflow.) In such applications, it is irresponsible programming to avoid error checking and irresponsible language design to encourage such avoidance.

We now turn to the question of how these contrasting approaches to arithmetic error may be described semantically. The semantics given in this chapter is easy to adopt to languages where arithmetic errors are not detected: One simply takes the arithmetic operations to be the erroneous functions actually computed by the hardware (which must of course be described precisely). The only restriction is that these operations must actually be functional. For example, \( x \div 0 \) must be some integer function of \( x \), and \( x + y \), even when the mathematically correct result would be unrepresentable, must be some integer function of \( x \) and \( y \). Thus, for example, the following equivalences would hold, regardless of the particular function denoted by division by zero or addition:

\[
\begin{align*}
\llbracket (x + y) \times 0 \rrbracket \textrm{intexp} \sigma &= 0 \\
\llbracket x \div 0 = x \div 0 \rrbracket \textrm{boolexp} \sigma &= \text{true} \\
\llbracket y := x \div 0 ; y := e \rrbracket \textrm{comm} \sigma &= \llbracket y := e \rrbracket \textrm{comm} \sigma \quad \text{when } y \notin \text{FV}_{\text{intexp}}(e) \\
\llbracket \text{if } x + y = z \text{ then } c \text{ else } c \rrbracket \textrm{comm} \sigma &= \llbracket c \rrbracket \textrm{comm} \sigma.
\end{align*}
\]

On the other hand, to treat the detection of arithmetic errors it is necessary to extend our present semantics in a nontrivial way, such as enlarging the set of results of integer and boolean expressions to include one or more special results denoting errors. We will take this approach to the denotational semantics of expressions when we consider functional languages in Section 11.6.

For the present, however, we will not consider error results for expressions. To do so would distract from our main concerns in studying imperative languages. It would also destroy the close coupling between the simple imperative language and predicate logic that underlies the approach to program specification and proof described in the next chapter.

2.8 Soundness and Full Abstraction

The goal of a denotational semantics is to abstract away from irrelevant details in order to focus on the aspects of a language that are of interest to the user of the semantics. Thus it is natural to ask whether one semantics is more abstract than another, and whether a semantics is excessively or insufficiently abstract for
the needs of the user. In fact, these seemingly intuitive questions can be asked precisely, for arbitrary semantic functions on arbitrary languages, as long as the semantics is compositional.

To avoid notational complications, we will limit our discussion to the semantics of a single kind of phrase; the specific example to keep in mind is the semantics of commands in the language of this chapter.

The easy question to formalize is when one semantics is more abstract than another:

- A semantic function $[-]_1$ is **at least as abstract** as a semantics function $[-]_0$ when, for all phrases $p$ and $p'$, $[p]_0 = [p']_0$ implies $[p]_1 = [p']_1$.

To go further, we formalize the “needs of the user” by assuming that there is a set of observable phrases and a set $O$ of observations, each of which is a function from the set of observable phrases to some set of outcomes.

To deal with compositionality, we define a **context** to be an observable phrase in which some subphrase has been replaced by a “hole” (which we denote by $-$). We write $C$ for the set of contexts. When $C$ is a context, we write $C[p]$ to denote the result of replacing (not substituting — there is no renaming) the hole in $C$ by the phrase $p$.

Then we formalize the idea that a semantics is not excessively abstract by

- A semantic function $[-]$ is **sound** if and only if, for any commands $c$ and $c'$,

$$[c] = [c'] \text{ implies } \forall O \in O. \forall C \in C. O(C[c]) = O(C[c']).$$

In other words, a semantics is sound if it never equates commands that, in some context, show observably different behavior.

Finally, we formalize the idea that a semantics is sufficiently abstract by

- A semantic function $[-]$ is **fully abstract** if and only if, for any commands $c$ and $c'$,

$$[c] = [c'] \text{ if and only if } \forall O \in O. \forall C \in C. O(C[c]) = O(C[c']).$$

In other words, a semantics is fully abstract if it distinguishes commands just when, in some context, they show observably different behavior. It is easy to see that a fully abstract semantics is a sound semantics that is at least as abstract as any sound semantics.

For the specific case of the semantics of the simple imperative language given in Section 2.2, suppose that all commands are observable and that an observation consists of starting the program in some state $\sigma$ and observing whether it terminates and, if so, what the value of some variable $v$ is. In other words, an observation is a function $O_{\sigma,v} \in (\text{comm}) \rightarrow \mathbb{Z}$ such that

$$O_{\sigma,v}(c) = \text{if } [c]_{\text{comm}}\sigma = \bot \text{ then } \bot \text{ else } [c]_{\text{comm}}\sigma v.$$
Then it is easy to see that \([c]_{\text{comm}}\) is fully abstract.

More surprisingly, suppose we limit observable commands to closed commands (which contain no occurrences of free variables) and only observe whether such commands terminate (which no longer depends on an initial state, since the commands are closed). Even with such restricted observations, our semantics remains fully abstract. For suppose \([c]_{\text{comm}} \neq [c']_{\text{comm}}\). Then there is a state \(\sigma\) such that \([c]_{\text{comm}}^\sigma \neq [c']_{\text{comm}}^\sigma\). Let \(\kappa_0, \ldots, \kappa_{n-1}\) be the variables that occur free in either \(c\) or \(c'\); let \(\kappa_0, \ldots, \kappa_{n-1}\) be constants whose values are \(\sigma v_0, \ldots, \sigma v_{n-1}\) respectively; and let \(C\) be the context

\[
C \overset{\text{def}}{=} \text{newvar } v_0 := \kappa_0 \text{ in } \cdots \text{newvar } v_{n-1} := \kappa_{n-1} \text{ in } -.
\]

Then, using the coincidence theorem (Proposition 2.6 in Section 2.5),

\[
[C[c]]_{\text{comm}}^\sigma_0 = [c]_{\text{comm}}[\sigma_0 | v_0 : \kappa_0 | \ldots | v_{n-1} : \kappa_{n-1}] = [c]_{\text{comm}}^\sigma
\]

\[
\neq [c']_{\text{comm}}^\sigma = [c']_{\text{comm}}[\sigma_0 | v_0 : \kappa_0 | \ldots | v_{n-1} : \kappa_{n-1}] = [C[c']]_{\text{comm}}^\sigma_0
\]

(where \(\sigma_0\) can be any state, since \(C[c]\) and \(C[c']\) are closed).

If, starting in \(\sigma\), one of the commands \(c\) or \(c'\) does not terminate, then they can be distinguished by observing \(C[c]\) and \(C[c']\). On the other hand, if both terminate, then there must be a variable \(v\) and a constant \(\kappa\) such that \([c]_{\text{assert}}^\sigma v = \kappa \neq [c']_{\text{assert}}^\sigma v\). In this case, the command can be distinguished by observing \(D[c]\) and \(D[c']\), where \(D\) is the context

\[
D \overset{\text{def}}{=} \text{newvar } v_0 := \kappa_0 \text{ in } \cdots \text{newvar } v_{n-1} := \kappa_{n-1} \text{ in } (- ; \text{if } v = \kappa \text{ then skip else while true do skip}).
\]

Although soundness and full abstraction are fundamental properties of semantics, it is vital to realize that they depend on the choice of what is observable, and also on the variety of contexts. For example, because our semantics takes the meaning of a command to be a function from initial states to final states, it abstracts away details of execution that, for some purposes, cannot be ignored. For example, the two commands

\[
x := x + 1 ; x := x + 1 \quad \text{and} \quad x := x + 2
\]

have the same meaning, as do

\[
x := 0 ; \text{while } x < 100 \text{ do } x := x + 1 \quad \text{and} \quad x := 100
\]

or

\[
x := x + 1 ; y := y \ast 2 \quad \text{and} \quad y := y \ast 2 ; x := x + 1,
\]

but there are situations where these pairs should not be considered equivalent. If our concept of observation included execution time or the continuous observation of program variables during execution, then we could distinguish each of the above pairs, so that our present semantics would be unsound.
Semantic definitions that make such distinctions are often called operational, in contrast to the denotational semantics developed in this chapter. Strictly speaking, however, these terms are relative — one can imagine a wide spectrum of definitions ranging from the extremely denotational to the extremely operational. (Some authors use “denotational” as a synonym for “compositional”. Occasionally it is difficult to tell which is meant, since in practice most denotational definitions are compositional and most operational definitions are not.)

A semantics can also become unsound if the variety of contexts is expanded by extending the language that is being defined. In Chapter 8 we will introduce a construction \( c_0 \parallel c_1 \) that causes two commands that may share variables to be executed concurrently. In this extended language, it is easy to construct contexts that will distinguish each of the above pairs. In Section 13.1, we will find that the interaction of assignment with a procedure mechanism can cause aliasing, so that assignment to \( x \) might change the value of \( y \) as well as \( x \). With this kind of language extension, one can define a context that distinguishes the last pair of commands. In both cases, we will need to move to a quite different semantics to avoid unsoundness.

**Bibliographic Notes**

The basic concept of a domain is due to Scott [1970; 1972], who originally defined a domain to be a complete, continuous, countably based lattice.

The more elementary and less restrictive concept of a domain used in this book is also described by Tennent [1991, Sections 5.2–5.3], Gunter [1992, Chapter 4], and Winskel [1993, Chapter 8]. (Note, however, that what we call predomains are called domains by Tennent and complete partial orders by Gunter and Winskel.)

The simple imperative language is discussed by Loeckx et al. [1987, Section 3.2]; domains and continuous functions are covered in Chapter 4 of the same book, and a denotational semantics for the language is given in Section 5.4.

The more advanced theory of domains, where the conditions of algebraicity and bounded completeness are imposed, is described in Gunter and Scott [1990] and Gunter [1992, Chapter 5]. An explanation of such domains in terms of the more concrete concept of an “information system” was given by Scott [1982], and is also described by Winskel [1993, Chapter 12].

A set of phrases defined by an abstract grammar, such as the set of commands of the simple imperative language, can be enlarged to form a domain by adding partial and infinite phrases appropriately. Moreover, any semantic function from the set of phrases to a domain, such as \([—]_{\text{comm}}\), has a unique continuous extension to such a domain of phrases. This fact can be used to give a purely syntactic description of loops and recursion by “unwinding” into infinite phrases. This idea was originally suggested by Scott [1971] and was pursued further by Reynolds [1977], and Goguen, Thatcher, Wagner, and Wright [1977].
Exercises

2.1 A double assignment command has the form $v_0, v_1 := e_0, e_1$. Its effect is to evaluate $e_0$ and $e_1$ and then assign the resulting values to $v_0$ and $v_1$ respectively. This differs from $v_0 := e_0 ; v_1 := e_1$ in that both expressions are evaluated before there is any state change.

Describe the syntax and semantics of this command by giving a production to be added to the abstract grammar in Section 2.1 and a semantic equation to be added to those in Section 2.2.

2.2 A repeat command has the form repeat $c$ until $b$. Its effect is described by the following flowchart:

```
enter --- c --- exit
```

(a) As in the previous problem, describe the syntax and semantics of this command by giving a production to be added to the grammar in Section 2.1 and a semantic equation to be added to those in Section 2.2. The semantic equation should express $\text{[repeat } c \text{ until } b\text{]}_{\text{comm}}$ as a least fixed-point of a function.

(b) Define the repeat command as syntactic sugar, as in Section 2.6.

(c) Prove that the definitions in parts (a) and (b) are equivalent.

2.3 Prove that

$$
\begin{cases}
[\text{while } x \neq 0 \text{ do } x := x - 2]_{\text{comm}} = \\
\{[\sigma | x:0] & \text{ if even}(\sigma x) \text{ and } \sigma x \geq 0 \\
\bot & \text{ if even}(\sigma x) \text{ and } \sigma x < 0 \\
\bot & \text{ if odd}(\sigma x).
\end{cases}
$$

2.4 Prove that the function $F$ in the semantic equation for the while command (Equation (2.4) in Section 2.4) is continuous.

2.5 Prove that, for all boolean expressions $b$ and all commands $c$,

$$
[\text{while } b \text{ do } c]_{\text{comm}} = [\text{while } b \text{ do } (c; \text{if } b \text{ then } c \text{ else skip})]_{\text{comm}}.
$$

2.6 Using the coincidence theorem for commands (Proposition 2.6 in Section 2.5), prove that, for all commands $c_0$ and $c_1$, if

$$
\text{FV}_{\text{comm}}(c_0) \cap \text{FA}(c_1) = \text{FA}(c_0) \cap \text{FV}_{\text{comm}}(c_1) = \{\},
$$

then

$$
[\text{[}c_0; c_1\text{]}]_{\text{comm}} = [\text{c_1; c_0]}_{\text{comm}}.
$$
2.7 In Section 2.5, we illustrated aliasing by giving a program for computing the factorial that becomes erroneous when the same variable is substituted for both the input variable \( x \) and the output variable \( y \). Rewrite this program so that it still works correctly when the same variable is substituted for \( x \) and \( y \).

2.8 Show that the substitution theorem for commands (Proposition 2.7 in Section 2.5) remains true when the condition on \( \delta \) is weakened from

\[
\delta w \neq \delta w' \text{ whenever } w \text{ and } w' \text{ are distinct members of } V
\]

to

\[
\delta w \neq \delta w' \text{ whenever } w \in FA(c), w' \in V, \text{ and } w \neq w'.
\]

2.9 The final version of the for command given in Section 2.6 has the defect that the value of the control value, after the last execution of the subcommand, is set to the next integer beyond the interval of iteration, which can cause unnecessary overflow. Define a for command (as syntactic sugar) that will never set the control variable to any integer outside the interval of iteration.

2.10 Consider introducing a \texttt{dotwise} construction, defined as syntactic sugar by

\[
\texttt{dotwise } c \overset{\text{def}}{=} c ; c.
\]

Explain why this is a valid definition, despite the fact that, when \( c \) contains two or more occurrences of \texttt{dotwise}, replacing \texttt{dotwise } \( c \) by \( c ; c \) will increase the number of occurrences of \texttt{dotwise}.