3 Type Systems and Semantics

"Ishmael: Surely all this is not without meaning."
Herman Melville, Moby Dick

CHAPTER OUTLINE

Type systems have become enormously important in language design because they can be used to formalize the definition of a language's data types and their proper usage in programs. Type systems are often associated with syntax, especially for languages whose programs are type checked at compile time. For these languages, a type system is a definitional extension that imposes specific syntactic constraints (such as the requirement that all variables referenced in a program be declared) that cannot be expressed in BNF or EBNF. For languages whose programs are type checked at run time, a type system can be viewed as part of the language's semantics. Thus, a language's type system stands at the bridge between syntax and semantics, and can be properly viewed in either realm.

The definition of a programming language is complete only when its semantics, as well as its syntax and type system, is fully defined. The *semantics* of a programming language is a definition of the *meaning* of any program that is syntactically valid from both the concrete syntax and the static type checking points of view.[1]

Program meaning can be defined in several different ways. A straightforward intuitive idea of program meaning is "whatever happens in a (real or model) computer when the program is executed." A precise characterization of this idea is called *operational semantics*.[2] Another way to view program meaning is to start with a formal specification of what the program is supposed to do, and then rigorously prove that the program does that by using a systematic series of logical steps. This approach evokes the idea of *axiomatic semantics.* A third way to view the semantics of a programming language is to define the meaning of each type of statement that occurs in the (abstract) syntax as a state-transforming mathematical function. Thus, the meaning of a program can be expressed as a collection of functions operating on the program state. This approach is called *denotational semantics.*

All three semantic definition methods have advantages and disadvantages. Operational semantics has the advantage of representing program meaning directly in the code of a real (or simulated) machine. But this is also a potential weakness, since defining the semantics of a programming language on the basis of any particular architecture, whether it be real or abstract, confines the utility of that definition for compiler-writers and programmers working with different architectures. Moreover, the virtual machine on which instructions execute also needs a semantic description, which adds complexity and can lead to circular definitions.

Axiomatic semantics is particularly useful in the exploration of formal properties of programs. Programmers who must write provably correct programs from a precise set of specifications are particularly well-served by this semantic style. Denotational semantics is valuable because its functional style brings the semantic definition of a language to a high level of mathematical precision. Through it, language designers obtain a functional definition of the meaning of each language construct that is independent of any particular machine architecture.

In this chapter, we introduce a formal approach to the definition of a language's type system. We also introduce the three models of semantic definition, paying special attention to the denotational model. Denotational semantics is used in later chapters for discussing various concepts in language design and for clarifying various semantic issues. This model is particularly valuable because it also allows us to actively explore these language design concepts in a laboratory setting.

---

1. Not too long ago, it was possible to write a syntactically correct program in a particular language that would behave differently when run on different platforms (with the same input). This situation arose because the definition of the language's semantics was not precise enough to require that all of its compilers translate a program to logically equivalent machine language versions. Language designers have realized in recent years that a formal treatment of semantics is as important as the formal treatment of syntax in ensuring that a particular program "means" the same thing regardless of the platform on which it runs. Modern languages are much better in this regard than older languages.

2. Technically, there are two kinds of operational semantics, called "traditional" and "structured" operational semantics (sometimes called "natural semantics"). In this chapter, we discuss the latter.

3.4 AXIOMATIC SEMANTICS

While it is important to programmers and compiler-writers to understand what a program does in all circumstances, it is also important for programmers to be able to confirm, or *prove,* that it does what it is supposed to do under all circumstances. That is, if someone presents the programmer with a specification for what a program is supposed to do, the programmer may need to be able to prove, beyond a reasonable doubt, that the program and this specification are absolutely in agreement with each other. That is, the program is "correct" in some convincing way. *Axiomatic semantics* provides a vehicle for developing such proofs.

For instance, suppose we want to prove mathematically that the C/C++ function *Max* in Figure 3.1 actually computes as its result the maximum of its two parameters: *a* and *b.*

Figure 3.1 A C/C++ Max Function

```
int Max (int a, int b) {
  int m;
  if (a >= b)
    m = a;
  else
    m = b;
  return m;
}
```

Calling this function one time will obtain an answer for a particular *a* and *b*, such as 8 and 13. But the parameters *a* and *b* define a wide range of integers, so calling it several times with all the different values to prove its correctness would be an infeasible task.

Axiomatic semantics provides a vehicle for reasoning about programs and their computations. This allows programmers to predict a program's behavior in a more circumspect and convincing way than running the program several times using as test cases different random choices of input values.

3.4.1 Fundamental Concepts

Axiomatic semantics is based on the notion of an *assertion,* which is a predicate that describes the *state* of a program at any point during its execution. An assertion can define the meaning of a computation, like "*the maximum of a and b*," without concern for how that computation is accomplished. For instance, the code in Figure 3.1 is just one way of algorithmically expressing the maximum computation; even for a function this

simple, there are many minor variations. In any case, the following assertion Q describes the function *Max* declaratively, without regard for the underlying computational process:

$Q \equiv m = max(a, b)$

This predicate defines the meaning of the function $Max(a, b)$ for any integer values of a and b. To prove that the program in Figure 3.1 actually computes $Max(a, b)$, we need to show that the logical expression Q is somehow equivalent in meaning to that program. Q is called a *postcondition* for the program *Max.*

Axiomatic semantics allows us to logically derive a series of predicates by reasoning about the behavior of each individual statement in the program, beginning with the postcondition Q and the last statement and working backwards. The last predicate, say P, that is derived in this series of steps is called the program's *precondition.* The precondition thus expresses what must be *true* before program execution begins in order for the postcondition to be satisfied.

In the case of *Max,* the postcondition Q is well-defined for all integer values of a and b. This suggests the following precondition:

$P \equiv true$

That is, for the program to be considered for correctness at all, no assumption or precondition is needed.

One final consideration must be taken into account before we look at the details of correctness proofs themselves. That is, for *some* initial values of the variables that satisfy the program's precondition P, executing the program may *never* reach its last statement. This may occur either because the program enters an infinite loop or because it may try to do a calculation that exceeds the capabilities of the machine on which it is running. For example, if we try to compute $n!$ for a large enough value of n,[8] the program will raise an arithmetic overflow condition and halt, thus never reaching its final goal. In general, we must be content to prove correctness of programs only partially—that is, only for those selections of initial values of variables that allow the execution of all its statements to be completed. This notion is called *partial correctness*.

These concerns notwithstanding, we can prove the (partial) correctness of a program by placing its precondition in front of its first statement and its postcondition after its last statement, and then systematically deriving a series of valid predicates as we simulate the execution of each instruction in its turn. For any statement or series of statements s, the following expression

$\{P\}s\{Q\}$

represents the predicate that the statement s is partially correct with respect to the precondition P and the postcondition Q. The expression $\{P\}s\{Q\}$ is called a *Hoare triple*[9] and reads "execution of statements s, beginning in a state that satisfies P, results in a state that satisfies Q, provided that s halts."

8. 30! exceeds the size of a 32-bit integer.

9. These forms are called *Hoare triples* since they were first characterized by C.A.R. Hoare in the original proposal for axiomatizing the semantics of programming languages [Hoare 1969].

For our example program, we first write the following Hoare triple:

$\{true\}$

```
if (a >= b)
  m = a;
else
  m = b;
```

$\{m = max(a, b)\}$

To prove the validity of this Hoare triple, we derive intermediate Hoare triples $\{P\}s\{Q\}$ that are valid for individual statements s, beginning with the postcondition. This process continues until we have successfully shown that the above Hoare triple is *true* for all initial values of a and b for which the program halts. That is, if we can derive Hoare triples that logically connect the individual lines in a program with each other, we will effectively connect the program's postcondition with its precondition.

How are these intermediate triples derived? That is done by formalizing what we know about the behavior of each type of statement in the language. Programs in Jay-like languages have four basic types of statements: assignments, conditionals, loops, and blocks (sequences). Each statement type has a *proof rule* which defines the meaning of that statement type in terms of the pre- and postconditions that it satisfies. The proof rules for Jay-like languages are shown in Table 3.1.

Table 3.1

Proof Rules for Different Types of Jay Statements

Statement Type	Proof Rule
1. *Assignment* s = s.target = s.source;	$\dfrac{true}{\{Q[\text{s.target}\backslash\text{s.source}]\}s\{Q\}}$
2. Sequence (*Block*) $s = s_1\ s_2$	$\dfrac{\{P\}s_1\{R\} \qquad \{R\}s_2\{Q\}}{\{P\}s_1\ s_2\{Q\}}$
3. *Conditional* s = if (s.test) s.thenpart else s.elsepart	$\dfrac{\{\text{s.test} \wedge P\}\text{s.thenpart}\{Q\} \qquad \{\neg\text{s.test} \wedge P\}\text{s.elsepart}\{Q\}}{\{P\}s\{Q\}}$
4. *Loop* s = while (s.test) s.body	$\dfrac{\{\text{s.test} \wedge P\}\text{s.body}\{P\}}{\{P\}s\{\neg\text{s.test} \wedge P\}}$
5. Rule of consequence	$\dfrac{P \supset P' \qquad \{P'\}s\{Q'\} \qquad Q' \supset Q}{\{P\}s\{Q\}}$

These proof rules are all of the form $\dfrac{premise}{conclusion}$, which is similar to the execution rules used in operational semantics. However, proof rules are to be read, "if the *premise* is valid then the *conclusion* is valid."

The *Assignment* proof rule has *true* as its premise, guaranteeing that we can always derive the conclusion; such a rule is called an *axiom.* The notation $Q[a\backslash b]$ means "the state that results from replacing the value of b in Q by a." For instance, if

$\{Q\} \equiv \{x = 1 \wedge y = 4\}$ then $\{Q[1\backslash x]\} \equiv \{1 = 1 \wedge y = 4\}$. Applied to an assignment, rule 1 suggests that the following Hoare triple is valid:

$\{a = max(a, b)\}$
```
  m = a;
```
$(m = max(a, b)\}$

That is, the replacement of m in the postcondition by a results in precondition of $\{a = max(a, b)\}$. So the assignment rule allows us to reason backwards through a program, deriving preconditions from postconditions in individual statements.

Proof rule 5 allows us to perform arithmetic and logical simplification in a predicate during the proof process. In the above example, for instance, the assertion $\{a \geq b \wedge a = max(a, b)\}$ is implied by $\{a \geq b\}$, so that we can substitute it and form an equivalent Hoare triple using rule 5 as follows:

$$\frac{a \geq b \supset a = max(a, b) \qquad \{a \geq b \wedge a = max(a, b)\}\text{m = a;}\{\text{m} = max(a, b)\}}{\{a \geq b\}m = \text{a;}\{\text{m} = max(a, b)\}}$$

This example also suggests that there may be several alternative preconditions that can be derived from a given statement and postcondition, using the proof rules. That precondition which is the least restrictive on the variables in play is called the *weakest precondition.* For instance, the precondition $\{a \geq b\}$ is the weakest precondition for the statement `m = a;` and its postcondition $\{m = max(a, b)\}$. Finding weakest preconditions is important because it enables simplification of the proof at various stages.

A strategy for proving the partial correctness of the rest of the program in Figure 3.1 works systematically from the postcondition backwards through the if, and then through the assignment statements in the then and else parts toward the given precondition.

Next, using the rule 1 for assignments and our postcondition on the else part of the if statement, we obtain:

$\{b = max(a, b)\}$
```
  m = b;
```
$\{m = max(a, b)\}$

As before, we use $a < b$ and rule 5 to show:

$$\frac{a < b \supset b = max(a, b) \qquad \{a < b \wedge b = max(a, b)\}\text{m = b;}\{\text{m} = max(a, b)\}}{\{a < b\}m = \text{b;}\{\text{m} = max(a, b)\}}$$

Having proved both premises of rule 3 for a conditional, we can conclude:

$\{true\}$
```
  if (a >= b)
    m = a;
  else
    m = b;
```
$\{m = max(a, b)\}$

In Subsection 3.4.2, we prove the correctness of a program involving a loop.

3.4.2 Correctness of Factorial

Suppose we want to prove mathematically that the C/C++ function *Factorial* in Figure 3.2 actually computes as its result $n!$, for any integer n where $n \geq 1$. By $n!$ we mean the product $1 \times \cdots \times n$.

Figure 3.2 A C/C++ Factorial Function

```
int Factorial (int n) {
  int f = 1;
  int i = 1;
  while (i < n) {
    i = i + 1;
    f = f * i;
  }
  return f;
}
```

So the precondition or input assertion Q for *Factorial* is $1 \leq n$, while the postcondition or output assertion is $f = n!$. In general in a program involving a loop, rule 4 of Table 3.1 is used to break the code into three parts, as follows:

```
{Q}
initialization
{P}
while (test ) {
   loopBody
}
```
$\{\neg \text{test} \wedge P\}$
```
finalization
{R}
```

where Q is our input assertion, R is our final or output assertion, and P is the loop *invariant*. An invariant is an assertion that remains *true* for every iteration of the loop. In general, there is no algorithmic way to derive a loop invariant from the input-output assertions for the program.[10] Thus, it is important for proving program correctness that the loop invariant be supplied for each loop.

For the *Factorial* function given in Figure 3.2, the loop invariant P is $\{1 \leq i \wedge i \leq n \wedge (f = i!)\}$. Thus, the *Factorial* program with its input-output assertions and loop invariant can be rewritten as:

$\{1 \leq n\}$
```
  f = 1;
  i = 1;
```
$\{1 \leq i \wedge i \leq n \wedge (f = i!)\}$

10. Finding a loop invariant is often tricky, as is the whole correctness proof process itself. Interested readers are encouraged to find additional sources (e.g., [Gries 1981]) that cover this very interesting topic in more detail.

```
  while (i < n) {
    i = i + 1;
    f = f * i;
  }
```
$\{i \geq n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!)\}$
$\{f = n!\}$
```
  return f;
```

This effectively reduces the original problem of proving the correctness of the original program to the three problems: (1) proving the initialization part; (2) proving the premise of rule 4; and (3) proving the finalization part. These subproblems may be proved in any convenient order.

The third part seems easiest, since it involves the empty statement (or the skip statement in Jay). The proof can be achieved by repeated applications of rule 5:

$$i \geq n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset (i = n) \wedge (f = i!) \supset (f = n!)$$

Since $i \geq n$ and $i \leq n$, it follows that $i = n$. The second step involves a substitution of one variable for another.

A strategy for proving the initialization part is to use rule 2 to break a *Block,* or sequence of statements, into its individual components:

$\{1 \leq n\}$
```
  f = 1;
```
{R'}
```
  i = 1;
```
$\{1 \leq i \wedge i \leq n \wedge (f = i!)\}$

The intermediate assertion R′ can be found from the application of rule 1 by back substitution: $\{1 \leq 1 \wedge 1 \leq n \wedge (f = 1!)\}$. Again, applying rule 1 to R′ on the program fragment:

$\{1 \leq n\}$
```
  f = 1;
```
$\{1 \leq 1 \wedge 1 \leq n \wedge (f = 1!)\}$

we obtain: $\{1 \leq 1 \wedge 1 \leq n \wedge (1 = 1!)\}$, which simplifies to $1 \leq n$. Thus, we have shown:

$$\frac{\{1 \leq n\}\text{f = 1;}\{1 \leq 1 \wedge 1 \leq n \wedge f = 1!\} \quad \{1 \leq 1 \wedge 1 \leq n \wedge f = 1!\}i = 1;\{1 \leq i \wedge i \leq n \wedge f = i!\}}{\{1 \leq n\}\ \text{f = 1; i = 1; } \{1 \leq i \wedge i \leq n \wedge f = i!\}}$$

Thus, it only remains to show that P is a loop invariant; this means that we must show the premise of rule 4, namely, that the loop body preserves the truth of the loop invariant:

$\frac{\{s.\text{test} \wedge P\}s.\text{body}\{P\}}{\{P\}s\{\neg s.\text{test} \wedge P\}}$, where `s` is a loop statement.

Specifically, we must show it for the given loop test, invariant P, and loop body:

$\{i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!)\}$

```
    i = i + 1;
    f = f * i;
```

$\{1 \leq i \wedge i \leq n \wedge (f = i!)\}$

Again we employ the strategy of using rule 2 for sequences to obtain:

$\{i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!)\}$

```
    i = i + 1;
{R'}
    f = f * i;
```

$\{1 \leq i \wedge i \leq n \wedge (f = i!)\}$

and then applying rule 1 on the assignment to f to find R′ by back substitution: $1 \leq i \wedge i \leq n \wedge (f \times i = i!)$. We repeat this strategy using R′ and rule 1 on the assignment to i to find: $1 \leq i \wedge i \leq n \wedge (f \times (i + 1) = (i + 1)!)$. If we can show that

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset 1 \leq i + 1 \wedge i + 1 \leq n \wedge (f \times (i + 1) = (i + 1)!),$$

then our proof is complete by rule 5. Using the following rule from logic:

$$\frac{p \supset q \qquad p \supset r}{p \supset q \wedge r}$$

and rule 5, we shall prove each term of the consequent separately. First, we must show:

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset 1 \leq i + 1,$$

This follows since $1 \leq i$ is a term in the antecedent. Next we must show:

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset i + 1 \leq n,$$

Again, since $i < n$ is in the antecedent and since we are dealing with integers, it follows that $i + 1 \leq n$. Finally, we must show:

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset (f \times (i + 1) = (i + 1)!),$$

Since $1 \leq i$, we can safely divide both sides of $f \times (i + 1) = (i + 1)!$ by $i + 1$, resulting in:

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset (f = i!),$$

which follows since the consequent appears as a term in the antecedent.

This concludes our proof of the (partial) correctness of the *Factorial* function given in Figure 3.2.

3.4.3 Correctness of Fibonacci

Suppose we want to prove mathematically that the C/C++ function *Fib* in Figure 3.3 actually computes as its result the nth Fibonacci number in the series 0, 1, 1, 2, 3, 5, 8, 13, 21, . . . , for any particular nonnegative value of n.

A strategy for proving the partial correctness of the rest of the program in Figure 3.3 works systematically from the postcondition backwards through the if, and then through the assignment statements in the then and else parts toward the given precondition.

Figure 3.3 A C/C++ Fibonacci Function

```
int Fib (int n) {
  int fib0 = 0;
  int fib1 = 1;
  int k = n;
  while (k > 0) {
    int temp = fib0;
    fib0 = fib1;
    fib1 = fib0 + temp;
    k = k - 1;
  }
  return fib0;
}
```

Rule 2 allows us to break a *Block,* or sequence of statements, into its individual constituents, and rules 3 and 4 allow us to reason through *Conditional* and *Loop* statements to derive their preconditions from their postconditions. As noted above, rule 5 allows us to cast off extraneous information along the way. Rule 5 is also useful when we want to introduce additional information in a predicate to anticipate what may be needed later in the process.

Now we can apply rule 1 two more times to derive a Hoare triple for each of the first three statements in the program:

$\{n \geq 0\}$
```
  fib0 = 0;
```
$\{n \geq 0 \wedge \text{fib0} = 0\}$

$\{n \geq 0 \wedge \text{fib0} = 0\}$
```
  fib1 = 1;
```
$\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1\}$

$\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1\}$
```
  k = n;
```
$\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1 \wedge 0 \leq k = n\}$

Now rule 2 gives us the ability to simplify this. For instance, if s_1 and s_2 are the first two statements in the above fragment, then:

$$\frac{\{n \geq 0\}s_1\{n \geq 0 \wedge \text{fib0} = 0\} \qquad \{n \geq 0 \wedge \text{fib0} = 0\}s_2\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1\}}{\{n \geq 0\}s_1\, s_2\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1\}}$$

allows us to rewrite the first two of these three Hoare triples as follows:

$\{n \geq 0\}$
```
  fib0 = 0;
  fib1 = 1;
```
$\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1\}$

Using rule 2 again allows us to establish the validity of the following Hoare triple:

$\{n \geq 0\}$
```
  fib0 = 0;
```

```
fib1 = 1;
k = n;
```
$\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1 \wedge 0 \leq k = n\}$

Next we consider the while loop. To deal with it, we need to discover an assertion that is true just before every repetition of the loop, including the first. That is, we need to find the loop's *invariant.* Finding the loop's invariant requires anticipation of what should be true when the loop finally terminates. Here is an invariant for our while loop:

$$INV = \left\{ \begin{array}{c} 0 \leq k \leq n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \ldots, n-k+1\}: Fib(j) = Fib(j-1) + Fib(j-2) \wedge \\ \text{fib0} = Fib(n-k) \wedge \text{fib1} = Fib(n-k+1) \end{array} \right\}$$

This says that for every value of k in which the loop has already run, we will have computed the Fibonacci numbers $Fib(0), Fib(1), \ldots, Fib(n-k+1)$, and the variables `fib0` and `fib1` are identical to the Fibonacci numbers $Fib(n-k)$ and $Fib(n-k+1)$ respectively. In particular, just before the loop begins we have $k = n$ (i.e., the loop will have computed no numbers), the value of `fib0` is $Fib(0)$, or 0, and the value of `fib1` is $Fib(1)$, or 1.

Note that this invariant is implied by the postcondition we carefully derived for the first three statements in the program. Thus, proof rule 5 allows the invariant to be valid before the first iteration of the loop begins. Looking ahead, we anticipate that when $k = n - 1$ the loop will have just computed $Fib(2)$ and the value of `fib0` will become $Fib(1) = 1$. When $k = n - 2$, the loop will have computed $Fib(2)$ and $Fib(3)$, leaving `fib0` $= Fib(2)$, and so forth.

This process continues until $k > 0$ is no longer true (i.e., $k = 0$), and the invariant asserts that the loop will have computed $Fib(2), Fib(3), \ldots,$ and $Fib(n+1)$, leaving the value of `fib0` $= Fib(n)$. Using the proof rule $\dfrac{\{s.\text{test} \wedge P\}s.\text{body}\{P\}}{\{P\}s\{\neg s.\text{test} \wedge P\}}$ in Table 3.1 for loop s, we relate the loop invariant with the precondition $s.\text{test} \wedge P$. Thus, when the loop terminates the test condition $k > 0$ is no longer *true,* which forces the following Hoare triple to be valid:

$$\left\{ \begin{array}{c} Fib(0) = 0 \wedge Fib(1) = 1 \wedge k = 0 \wedge \\ \forall j \in \{2, \ldots, n-0+1\}: Fib(j) = Fib(j-1) + Fib(j-2) \wedge \\ \text{fib0} = Fib(n-0) \wedge \text{fib1} = Fib(n-0+1) \end{array} \right\}$$

This logically implies the program's postcondition, using proof rule 5 (the rule of consequence) along with some fairly obvious simplifications:

$$\{Fib(0) = 0 \wedge Fib(1) = 1 \wedge \forall j \in \{2, \ldots, n\}: Fib(j) = Fib(j-1) + Fib(j-2) \wedge \text{fib0} = Fib(n)\}$$

A final task is to show that the invariant *INV* does remain valid for each repetition of the statements inside the body of the loop. That is, we need to show that the following (ugly!) Hoare triple is valid for every value of $k > 0$:

$$INV = \left\{ \begin{array}{c} 0 \leq k-1 < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \ldots, n-(k-1)\}: Fib(j) = Fib(j-1) + Fib(j-2) \wedge \\ \text{fib0} = Fib(n-k) \wedge \text{fib1} = Fib(n-(k-1)) \end{array} \right\}$$

```
int temp = fib0;
fib0 = fib1;
fib1 = fib0 + temp;
k = k - 1;
```

$$INV' = \left\{ \begin{array}{c} 0 \leq k < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \ldots, n-k\}: Fib(j) = Fib(j-1) + Fib(j-2) \wedge \\ \text{fib0} = Fib(n-k) \wedge \text{fib1} = Fib(n-(k-1)) \end{array} \right\}$$

To accomplish this, we use proof rule 5 to simplify intermediate expressions, and substitute equivalent expressions using our knowledge of algebra. For instance, we use the facts that the expression $n - k + 1$ in the above invariant *INV* is equivalent to $n - (k - 1)$, and $k > 0$ in the invariant is equivalent to $k - 1 \geq 0$.

Looking at the intermediate statements inside a single iteration of the loop, we see that they maintain the validity of the invariant and make progress toward the final computation of `fib0` $= Fib(n)$.

$$INV = \left\{ \begin{array}{c} 0 \leq k-1 < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \ldots, n-(k-1)\}: Fib(j) = Fib(j-1) + Fib(j-2) \wedge \\ \text{fib0} = Fib(n-k) \wedge \text{fib1} = Fib(n-(k-1)) \end{array} \right\}$$

```
int temp = fib0;
```

$$\left\{ \begin{array}{c} 0 \leq k-1 < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \ldots, n-(k-1)\}: Fib(j) = Fib(j-1) + Fib(j-2) \wedge \\ \underline{\text{fib0} = Fib(n-k) \wedge \text{fib1} = Fib(n-(k-1)) \wedge \text{temp} = Fib(n-k)} \end{array} \right\}$$

```
fib0 = fib1;
```

$$\left\{ \begin{array}{c} 0 \leq k-1 < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \ldots, n-(k-1)\}: Fib(j) = Fib(j-1) + Fib(j-2) \wedge \\ \underline{\text{fib0} = Fib(n-(k-1)) \wedge \text{fib1} = Fib(n-(k-1)) \wedge \text{temp} = Fib(n-k)} \end{array} \right\}$$

```
fib1 = fib0 + temp;
```

$$\left\{ \begin{array}{c} 0 \leq k-1 < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \ldots, n-(k-1)\}: Fib(j) = Fib(j-1) + Fib(j-2) \wedge \\ \underline{\text{fib0} = Fib(n-(k-1)) \wedge \text{fib1} = Fib(n-(k-1)) + Fib(n-k) \wedge \text{temp} = Fib(n-k)} \end{array} \right\}$$

```
k = k - 1;
```

$$\left\{\begin{array}{c} 0 \le \underline{k} < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \ldots, \underline{n-k}\}: Fib(j) = Fib(j-1) + Fib(j-2) \wedge \\ \underline{\text{fib0} = Fib(n-k) \wedge \text{fib1} = Fib(n-(k-1)) \wedge \text{temp} = Fib(n-k)} \end{array}\right\}$$

To clarify the above, we have underlined that part of the invariant that changes after each of these four statements is taken into account. The last line implies the transformation of the invariant to the following form; since `temp` is a variable to be reassigned at the beginning of the next loop iteration, it can be dropped from the final assertion. Thus, the invariant INV′ at the end of a single iteration is transformed as follows:

$$INV' = \left\{\begin{array}{c} 0 \le k < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \ldots, n-k\}: Fib(j) = Fib(j-1) + Fib(j-2) \wedge \\ \text{fib0} = Fib(n-k) \wedge \text{fib1} = Fib(n-(k-1)) \end{array}\right\}$$

The key here is to notice that the values of `fib0` and `fib1` are transformed so that they represent the next adjacent pair of Fibonacci numbers following the pair that they represented at the beginning of this sequence.

The indexing is a bit tricky, but readers should see that the three Fibonacci numbers that influence a single iteration are $Fib(n-k)$, $Fib(n-(k-1))$, and $Fib(n-(k-2))$, in ascending order. For instance, when $k = n$, this group of statements begins with `fib0` $= Fib(0)$ and `fib1` $= Fib(1)$. This group of statements ends with $k = n-1$, `fib0` $= Fib(n-(n-1)) = Fib(1)$, and `fib1` $= Fib(n-((n-1)-1)) = Fib(2)$.

3.4.4 Perspective

Axiomatic semantics and the corresponding techniques for proving the correctness of imperative programs were developed in the late 1960s and early 1970s. At that time it was the expectation that by now most programs would routinely be proven correct. Clearly that has not happened.

Actually, the importance of correctness proofs in software design has been a subject of heated debate, especially in the early 1990s. Many software engineers reject the notion of formal proof [DeMillo 1979], arguing that it is too complex and time-consuming a process for most programmers to master. Instead they use elaborate testing methods to convince themselves that the software runs correctly most of the time.

The counter argument was made by Dijkstra [1972] who stated that testing could only prove the presence of bugs, never their absence. Consider a simple program that inputs two 32-bit integers, computes some function, and outputs a 32-bit integer. There are 2^{64} possible inputs (approximately 10^{20}), so that even if one could test and verify (!) 100 million test cases per second, complete testing would take approximately 10^5 years. And this is for one of the simplest programs one can imagine!

Is there a middle ground between complex and time-consuming proofs and totally inadequate testing? We believe so.

First, properties of programs other than correctness can be routinely proved. These include safety of programs where safety is a critical issue. Absence of deadlock in concurrent programs is also often formally proved.

Second, the methods in object-oriented programs (as we shall see in Chapter 7) are often quite small in size. Informal proofs of such methods are routinely possible, although not often practiced. One reason for this is that many programmers, largely ignorant of formal program correctness, are not able to precisely state the input-output assertions for the methods they write.

Third, programmers trained in program correctness can and do state input-output assertions for the methods they write using formal English (or other natural language); this leads to vastly improved documentation.

As an example where such formalism could (and should) have been used, consider Sun's `javadoc` documentation for the various `String` methods in JDK 1.1. The comment for the method:

```
public String substring(int beginIndex, int endIndex);
```

states: "Returns a new string that is a substring of this string." How imprecise! How would an implementor carry out an informal proof given such a vague specification? What are the range of valid values for `beginIndex` and `endIndex`? Is the minimum valid value for `beginIndex` 0 or 1? A programmer interested in producing an informal proof of an implementation of `substring` would at least require a more formal description of this method; such a description is left as an exercise.

3.5 DENOTATIONAL SEMANTICS

The *denotational semantics* of a language defines the meanings of abstract language elements as a collection of environment- and state-transforming functions. The *environment* of a program is the set of objects and types that are active at each step during its execution. The *state* of a program is the set of all active objects and their current values. These state-transforming functions depend on the assumption of some primitive types and transformations.

While axiomatic semantics is valuable for clarifying the meaning of a program as an abstract text, operational (or denotational) semantics addresses the meaning of a program as an active object within a computational (or functional) environment. The use of denotational semantics for defining meaning has both advantages and disadvantages. An advantage is that we can use the functional denotations of program meaning as the basis for specifying an interpreter for the language. However, this advantage raises an additional issue. That is, strictly speaking, the use of Java to implement the functional definition of, say, a *Loop* requires that we define the semantics of Java *itself* before using it to implement the semantics of Jay. That is, there is some circularity in using the denotational model as a basis for defining the meaning of a language.

Nevertheless, denotational semantics is widely used, and it allows us to define the meaning of an abstract Jay program as a series of state transformations resulting from the application of a series of functions *M*. These functions individually define the meaning of every class of element that can occur in the program's abstract syntax tree—*Program, Block, Conditional, Loop, Assignment,* and so forth.

Let Σ represent the set of all program states σ. Then a meaning function *M* is a mapping from a particular member of a given abstract class and current state in Σ to a new

EXERCISES

3.1 Expand the static type checking function V for *Declaration*s so that it defines the requirement that the type of each variable be taken from a small set of available types, say {int, boolean}. Use the same functional style and abstract syntax for *Declaration*s that are discussed in this chapter.

3.2 Expand the Java method that implements the function V for *Declaration*s so that it implements the additional requirement stated in Question 3.1.

3.3 Argue that the Java method that implements the function V for *Declaration*s is correct, in the sense that it covers all the cases that the function itself covers.

3.4 Suppose $\sigma_1 = \{\langle x, 1\rangle, \langle y, 2\rangle, \langle z, 3\rangle\}$, $\sigma_2 = \{\langle y, 5\rangle\}$, and $\sigma_3 = \{\langle w, 1\rangle\}$. What are the results of the following operations?

(a) $\sigma_1 \overline{\cup} \sigma_2$
(b) $\sigma_1 \overline{\cup} \sigma_3$
(c) $\sigma_2 \overline{\cup} \sigma_3$
(d) $\varnothing \overline{\cup} \sigma_2$
(e) $\sigma_1 \otimes \sigma_3$
(f) $\sigma_2 \otimes \sigma_3$
(g) $(\sigma_1 - (\sigma_1 \otimes \sigma_3)) \cup \sigma_3$

3.5 Complete the operational semantics for the arithmetic, boolean, and logical operators of Jay by writing an execution rule for each operator.

3.6 Derive the complete operational semantics for the following program segment, showing all execution rule applications and deriving the final state that includes the result ⟨fib0, 3⟩. Assume that the initial state $\sigma = \varnothing$.

```
int fib0 = 0;
int fib1 = 1;
int k = 4;
while (k > 0) {
  int temp = fib0;
  fib0 = fib1;
  fib1 = fib0 + temp;
  k = k - 1;
}
```

3.7 Below is a Hoare triple that includes a Jay program segment to compute the product z of two integers x and y.

$\{y \geq 0\}$
```
z = 0;
n = y;
while (n>0) {
  z = z + x;
  n = n - 1;
}
```
$\{z = xy\}$

(a) What proof rules in Table 3.1 and additional knowledge about algebra can be used to infer that the precondition in this Hoare triple is equivalent to the assertion $\{y \geq 0 \wedge 0 = x(y - y)\}$?
(b) Using the assignment proof rule, complete the following Hoare triple for the first two statements in this program:

$\{y \geq 0 \wedge 0 = x(y - y)\}$
```
z = 0;
n = y;
```
$\{y \geq 0 \wedge \qquad\qquad\qquad \}$

(c) Explain how the following can be an invariant for the while loop in this program.

$\{y \geq 0 \wedge n \geq 0 \wedge z = x(y - n)\}$

That is, why is this assertion *true* before execution of the first statement in the loop, and why must it be *true* before execution of every successive repetition of the loop?
(d) Show that this invariant holds for a single pass through the loop's statements.
(e) Using the proof rule for loops, show how the invariant is resolved to an assertion that implies the validity of the postcondition for the entire program.

3.8 Write a Jay program segment that computes the sum of a series of integers $a_0, \ldots, a_{n-1}$, given that $n \geq 0$. Write pre- and postconditions for this program, and then develop a proof of its correctness using the proof rules in Table 3.1.

3.9 Write appropriate pre- and postconditions for the method `substring(int,int)` in the class `java.lang.String`.

3.10 Write appropriate pre- and postconditions for the method `indexOf(String)` in the class `java.lang.String`.

3.11 Give a recursive C/C++ implementation of the function *Fib* in Figure 3.3. Prove the partial correctness of your recursive implementation for all values of $n \geq 0$.

3.12 A program has *total correctness* if it (completes its execution and) satisfies its postcondition for *all* input values specified in its precondition.
(a) Experimentally determine the largest value of n for which the function *Fib* in Figure 3.3 delivers a result. What happens when it does not?
(b) Redefine the precondition for *Fib* so that its correctness proof becomes a proof of total correctness.
(c) Revise its correctness proof so that it reflects this new precondition.

3.13 Consider the following sequence of C/C++ statements, which are syntactically valid but have no reasonable semantic interpretation (assuming that i and j have been declared as `int` variables):

```
j = 0;
i = 3/j;
for (i=1; i>-1; i++)
   i--;
```

How are these situations handled when executed by your C/C++ system?

3.14 Give other kinds (beyond those kinds illustrated in the previous question) of C/C++ statements that are syntactically valid but whose meaning cannot be reasonably defined in the semantics of a programming language.

3.15 (a) How does Java define the numerical idea of infinity? (You should look at the *Java Language Specification* [Gosling 1996] for the details.)
(b) Looking at the specifications in the *Java Language Definition,* can you explain in plain English the meaning of the statement i = 3 / j; for all possible values of j, including 0?
(c) (Optional) Can you write a functional definition for the meaning of division in Java using these ideas? Start with the prototype function $M : \mathit{Division} \times \Sigma \to \Sigma$.

3.16 Consider the expression `x + y/2` in the language C. How many different interpretations does this expression have, depending on the types of x and y. Can you find a language in which this expression can denote vector or matrix arithmetic, when x and y themselves denote vectors or matrices?

3.17 Show how the meaning of each of the following expressions and given states are derived from the functions *M* and *ApplyBinary* given in this chapter. (You developed the abstract syntax for each of these expression as an exercise in Chapter 2.)
(a) M(z+2)*y, {⟨x, 2⟩, ⟨y, −3⟩, ⟨z, 75⟩})
(b) M(2*x+3/y−4, {⟨x, 2⟩, ⟨y, −3⟩, ⟨z, 75⟩})
(c) M(1, {⟨x, 2⟩, ⟨y, −3⟩, ⟨z, 75⟩})

3.18 Show all steps in the derivation of the meaning of the following assignment statement when executed in the given state, using this chapter's definitions of the functions *M* and *ApplyBinary.*
M(z=2*x+3/y−4, {⟨x, 6⟩, ⟨y, −12⟩, ⟨z, 75⟩})