J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*

PROGRAMMING THEORY

The theory of programming consists of a body of theoretical techniques for reasoning about program behavior. This body of techniques is divided into two classes: (1) techniques for reasoning about the *functional behavior* of programs, and (2) techniques for reasoning about *performance issues* such as time and space complexity. For historical reasons, only the techniques of the first class are referred to as "theory of programming," while those of the second class are referred to as "algorithm analysis." The two sections of the journal *Theoretical Computer Science* represent this division.

Theory of programming includes a vast array of formal and semiformal disciplines that address a wide range of issues:

- *Program specification* addresses the issue of how to specify the intended functional behavior of programs. (See also Formal Specification of Software.)
- *Programming language semantics* addresses the issue of how programs behave. It studies mathematical models for capturing such behavior and techniques for reasoning about them.
- *Programming logic* studies reasoning principles for proving that programs have the intended behavior.
- *Program verification* builds on programming logic and studies practical techniques for proving program correctness. (See also Program Testing.)
- *Program derivation* studies techniques and formal rules using which programs can be derived from specifications. Automated techniques for doing such derivation go by the names of *program synthesis* and *program transformation*.
- *Formal methods* integrate all these techniques for practical software development. (See also Vienna Development Method and Software Prototyping.)

Since programming theory is such a broad area, we obtain focus in this article by concentrating on the issue of *functional correctness* of programs. This leads us through the sub-areas of program specification, programming semantics and programming logic, with an emphasis on the last subject. Other aspects of this area are discussed in the articles mentioned above.

Program Specification

For many algorithms that arise in practice, it is possible to state precisely what the algorithm is supposed to achieve. For example, a sorting algorithm is expected to rearrange the elements of a collection in the increasing (or decreasing) order. A compiler for a programming language must translate a program in the source language into one in the machine language with the same behavior. Formalizing this involves defining the "behavior" of programs in the source and machine languages. A theorem prover for some logical system is expected to say *yes* or *no* depending on whether the input is a theorem in the logical system. A database search engine must produce all and only those records in the database that form an answer to the input query. A suitably formalized version of such statements constitutes a *specification* for the algorithm and the fact that the algorithm meets the specification is called its *correctness*.

The reader can see that formalizing such specifications often involves its own theory, which we might call the *application domain theory*. Research papers that publish algorithms often develop such application domain theory in order to prove the correctness of the algorithms.

A second method of proving correctness involves the notion of program *equivalence*. Suppose we do not possess a good application domain theory to give a formal specification of a problem. We might nevertheless be able to write a naive program that can be clearly seen to be a correct solution. We can then show the correctness of a real program by proving that it is equivalent to the naive program. For example, an algorithm for finding paths in a directed graph can be proved correct by showing its equivalence with a naive program that computes the transitive closure of the graph using set-theoretic operations like union. The program equivalence method is especially appropriate for programs that involve sophisticated data structures, for programs that cache intermediate results in clever ways, and for parallel or concurrent algorithms.

The techniques that are surveyed in this article are routinely used for small algorithms and in research publications, but there is no clear agreement on how far they are applicable to real software systems. Software systems involve such a massive amount of detail that writing complete specifications for them is itself a challenging enterprise. Moreover, the application domain theories for various software applications are not well developed. Thus, the state of the art is very far from being able to prove the correctness of a real-life software system such as an air traffic control system or a telephone switching system. However, with the present state of the art, the following applications of programming theory are recommended:

- Specification Significant components of software systems can be formally specified. Such specifications allow a high degree of certainty for the implementors of the component as well as the client code. Often they also lead to clean interfaces between components.
- Algorithm and Data Structure Correctness The correctness verification techniques can be applied to small portions of systems that involve sophisticated algorithms and clever data structures. Examples include storage management modules, central data structure like symbol tables in compilers, scheduling algorithms, communication protocols, etc.
- Safety Properties It is often possible to identify critical "safety" properties that are necessary to avoid severe loss of life or property. Such properties can be specified and proved for an entire software system.

Notwithstanding the extent of application, we believe that programming theory is an indispensable part of every serious software professional's arsenal. A conscious attention to correctness concerns often leads to clear and well-organized code even if the verification techniques are not rigorously applied.

Proof Methods. Methods for proving properties of programs can be classified into three layers: *operational* methods, *denotational* methods, and *axiomatic* methods. These are not mutually exclusive classes, but form different levels of abstraction for the same basic ideas.

Operational methods are based on the notion of operational equivalence. Two program phrases P_1 and P_2 (such as procedures or abstract data types) are said to be operationally equivalent if, in all possible complete-program contexts, using P_1 and P_2 gives the same results. In other words, P_1 and P_2 must be interchangeable in all contexts. No sophisticated mathematical ideas are required to understand this notion. All that is needed is a clear idea of how complete programs behave. For example, a good abstract interpreter for the programming language suffices. However, using this definition in practice can prove quite tedious because one must check the condition for *all* program contexts.

Denotational methods are based on the notion of "meaning" in an appropriate denotational semantic model. A denotational model is a mathematical structure in which every program phrase can be interpreted. For example, it is common to interpret types as sets and programming functions as mathematical (set-theoretic) functions. Commands in imperative programming languages are often interpreted as functions from states to states, expressions as functions from states to values and so on. Given such a denotational model, we consider two program phrases as equivalent if they have the same denotation (meaning) in the model. Thus, denotational methods avoid having to consider all possible program contexts. They can also deal with various properties that can be stated for the denotations, not only equivalence. But the mathematical tools involved in the models can be quite sophisticated, especially if one wants good accuracy.

Axiomatic methods give formal rules for deducing certain kinds of program properties. One kind of property is just equivalence, but other kinds of properties are possible. An excellent example is that of *Hoare triples* $\{P\}C\{Q\}$, which can be used to reason about commands. The Hoare triple $\{P\}C\{Q\}$ says that, in any state in which the condition P holds, running the command C (possibly) gives a state in which Q holds. C. A. R. Hoare formulated an axiom system for deducing such properties for commands, which has been extremely successful in practical applications. One can apply axiomatic methods without any sophisticated mathematical background. However, it can be tedious to deduce facts carefully using the rules of an axiom system. Failure to check all the conditions properly can lead to incorrect results. Some knowledge of the underlying denotational model can be helpful in taking well-motivated leaps in reasoning.

The three classes of methods just mentioned above are closely related. The operational equivalence is a hard bound on possible equivalences because if two phrases are not operationally equivalent, then their difference is observable in some context. Thus, any equivalence provable by denotational or axiomatic methods must in fact be an operational equivalence. Second, the soundness of axiomatic methods is typically proved using a denotational model. Then any fact proved using the axiom system would hold in the model. Thus, the relation between provable facts using various methods are related as follows:

$$\text{axiomatic} \subseteq \text{denotational} \subseteq \text{operational}$$

Further discussion of the three approaches may be found in Ref. 1.

Programming Language Frameworks

The most widely used programming languages such as C, Ada, and Modula are called *imperative* programming languages. This is because the basic construct in these languages is the executable command. Another class of languages of interest to us is that of *functional* programming languages. Historically, the idea of functional programming arose from the observation that the procedure concept in programming is similar to that of mathematical functions (2). It was later recognized that most programs of interest can be expressed just using the function or procedure concept without any commands at all. (See also Functional programming.) Our interest in functional programming lies in the fact that most concepts of programming theory arise in functional programming in a simplified form. The theory of imperative programming can then be obtained by adding to functional programming the notion of commands.

Logic programming languages are closely related to functional programming and their programming theory is essentially similar. So, we will not treat this class separately. (See also Logic programming.)

Object-oriented programming is essentially concerned with data abstraction. We treat the issues of data abstraction in connection with both functional and imperative settings. (See also Abstract data types.)

The remainder of this article is organized under the headings:

Functional Programs
Abstract Data Types
Imperative Programs
Procedures and Objects

Under each heading, we discuss the theoretical techniques appropriate for that class of programs.

```
sort : [Int] → [Int]
sort []= []
sort x:xs = (sort lo)++[x]++(sort hi)
                  where (lo, hi) = partition (x, xs)
partition : (Int, [Int]) → ([Int], [Int])
partition (p, []) = ([], [])
partition (p, x:xs) = if x < p then
                         (x:lo, hi)
                      else
                         (lo, x:hi)
                            where (lo, hi) = partition (p, xs)
```

Fig. 1. Quick sort.

```
sort [2, 5, 1]
     = (sort lo) ++ [2] ++ (sort hi) where (lo, hi) = partition (2, [5, 1])
partition (2, [5, 1])
     = (lo, 5:hi) where (lo, hi) = partition (2, [1])
     = (lo, 5:hi) where (lo, hi) = (1:lo', hi') where (lo', hi') = partition (2, [])
     = (lo, 5:hi) where (lo, hi) = (1:[], [])
     = (1:[], 5:[])
     = ([1], [5])
sort [2, 5, 1]
     = (sort [1]) ++ [2] ++ (sort [5])
     = [1] ++ [2] ++ [5]
     = [1, 2, 5]
```

Fig. 2. Sample computation using sort.

Functional Programs

Functional programming best illustrates the direct use of denotational methods in reasoning about programs. Our treatment involves elementary mathematical concepts of sets and functions, and mathematical induction on natural numbers.

Figure 1 shows a sample program in the programming language Haskell (3) for sorting a list of integers using the quick-sort method. The function sort takes a list of integers as input and produces the sorted list of integers as output. (The first line gives the type of sort where the notation [Int] stands for the type "list of integers.") The function partition takes an integer p and a list of integers xs and produces a pair of lists (lo, hi) where lo contains the elements of xs that are less than or equal to p, and hi contains the remaining elements. The notation for lists is as follows: [] denotes the empty list and x:xs denotes the list obtained by adding x at the front of the list xs. So, a list with elements $x_1, \ldots, x_n$ is denoted in the Haskell notation as

$x_1{:}x_2{:}\ldots{:}x_n{:}[]$

Such a list can also be written as $[x_1, \ldots, x_n]$. The symbol ++ represents the list append function. The *where* clause allows one to define one or more variables via local definition. Computation proceeds by expanding function applications by their definitions and simplifications. See Fig. 2.

The denotational model we use interprets types as sets (Int is the set of integers, [Int] is the set of lists over integers, etc.) and functions as mathematical functions. Because the recursive calls are made for smaller lists, there are unique functions sort and partition that satisfy the equations in the program. Thus, we can treat sort and partition as ordinary mathematical functions and apply ordinary mathematical reasoning. We illustrate this by proving the correctness of sort.

Theorem 1. If sort xs = ys, then ys contains the same collection of elements as xs and is ordered.

We also need a lemma for the partition function.

Lemma 2. *If partition(p, xs) = (lo, hi), then*

(1) lo ++ hi has the same collection of elements as xs
(2) All the elements of lo are less than or equal to p ("small" values)
(3) All the elements of hi are greater than p ("large" values)

There is some ambiguity in these statements because we have not defined what is meant by having the *same collection of elements*. Fortunately, all we need are the following facts. Use the notation xs $\approx$ ys to mean xs and ys have the same collection of elements.

- The relation $\approx$ is an equivalence relation.
- If xs $\approx$ xs$'$ and ys $\approx$ ys$'$, then xs ++ ys $\approx$ xs$'$ ++ ys$'$.
- xs ++ ys $\approx$ ys ++ xs.
- If xs $\approx$ ys, then length(xs) = length(ys).

More formally, one interprets *collection* as the mathematical notion of multiset and, by induction, defines xs $\approx$ ys to mean that the multiset of elements of xs and ys are equal. The facts just noted are provable from this formalization.

Proof of Lemma 2. By induction on the length of xs:

- If the length is 0, that is, xs = [], then lo = [] and hi = [] and the statement clearly holds because xs = lo ++ hi.
- If the length is positive, let xs = x:xs$'$. Since xs$'$ is shorter than xs, the lemma holds for xs$'$ by induction. Hence, if partition(p, xs$'$) = (lo$'$, hi$'$), then lo$'$ ++ hi$'$ has the same elements as xs$'$, and lo$'$ contains "small" values and hi$'$ contains "large" values. If x $\leq$ p then (lo, hi) = (x:lo$'$, hi$'$). Clearly, lo ++ hi has the same elements as x:xs. The elements of lo = x:lo$'$ are "small" and those of hi = hi$'$ are "large." The case x $>$ p is similar.

Proof of Theorem 1. By induction on the length of xs.

- If the length is 0, that is, xs = [], then sort xs = [] and the statement clearly holds.
- If the length is positive, let xs = x:xs$'$ and partition(x, xs$'$) = (lo, hi). By lemma 2, we have that lo ++ hi has the same collection of elements as that of xs$'$ and hence has the same length as xs$'$. Since xs$'$ is shorter than xs, both lo and hi are shorter than xs. So, the inductive hypothesis applies to lo and hi, and sort lo and sort hi are sorted versions of lo and hi. It is easy to see that (sort lo) ++ [x] ++ (sort hi) satisfies the statement of the theorem.

The remarkable feature of the preceding correctness proof is that it directly encodes the informal reasoning programmers use in thinking about correctness of programs. No advanced mathematical theories or special logical notations are involved. Experienced functional programmers often carry out simple proofs like this mentally without writing down a single word. This leads to a high degree of reliability for functional programs.

```
reverse : [t] → [t]
reverse []= []
reverse (x:xs) = (reverse xs) ++[x]

rev : [t] → [t]
rev xs = loop (xs, [])
loop : ([t],[t]) → [t]
loop ([], p) = p
loop (x:xs, p) = loop(xs, x:p)
```

Fig. 3. Two programs for list reverse.

```
rev [1, 2, 3]
    = loop ([1, 2, 3], [])
    = loop ([2, 3], [1])
    = loop ([3], [2, 1])
    = loop ([], [3, 2, 1])
    = [3, 2, 1]
```

Fig. 4. Sample computation using rev.

Next, we consider a proof of program equivalence in which explicit manipulation of expressions will be involved. Figure 3 shows two programs for reversing a list. The first program reverse is a straightforward solution whose correctness is more or less obvious. But it is inefficient: it has $O(n^2)$ time complexity because the append operation ++ takes time linear in the length of its first argument. The second program rev has $O(n)$ complexity, but its correctness is far from obvious. In fact, unless the reader has experience with similar programs, it is hard to believe that it works at all. Figure 4 shows a simple computation that gives some insight into how rev works.

We would like to show the correctness of rev by proving that it is equivalent to reverse. The key to the proof is coming up with a lemma that captures the behavior of loop. The sample computation of Fig. 4 suggests that loop reverses its first argument and appends it to the front of the second argument. This insight leads to the following lemma.

Lemma 3. *loop(xs, p) = (reverse xs) ++ p*

Proof. By induction on the length of xs:

- loop([], p) = p = [] ++ p = (reverse []) ++ p
- loop(x:xs′, p)
  = loop(xs′, x:p) by definition of loop
  = (reverse xs′) ++ (x:p) by inductive hypothesis
  = (reverse xs′) ++ ([x] ++ p) by inductive hypothesis
  = ((reverse xs′) ++ [x] ++ p by associativity of ++
  = (reverse (x:xs′)) ++ p by definition of reverse

The correctness of rev is immediate from the lemma.

Theorem 4. rev xs = reverse xs

Proof. rev xs = loop(xs, []) = (reverse xs) ++ [] = reverse xs

Equational proofs of this kind arise commonly in verifying program optimizations and program restructuring. In fact, since equational reasoning steps are invertible, they can be used for program transformation. One starts with an unoptimized program and applies equational steps to derive an optimized version. Pioneered by Burstall and Darlington (4), the technique of program transformation is widely used by the functional programming community (5,6).

Type Theory. Functional programming is implicitly based on a type theory that is often referred to as *typed lambda calculus*. A type theory consists of a collection of types built from designated type constructors. For each type constructor, there are term-forming operations that build or unbuild values of the type and there are equations that specify that building and unbuilding cancel each other out. We illustrate this for two type constructors:

- Whenever $A_1, \ldots, A_n$ are types (for $n \geq 0$), there is a type $(A_1, \ldots, A_n)$ that we think of as the type of n-tuples (or the product type). The term-forming operations are as follows:

  (1) If $M_1{:}A_1, \ldots, M_n{:}A_n$ are terms of their respective types in some context, then the term $(M_1, \ldots, M_n)$ is a term of type $(A_1, \ldots, A_n)$.
  (2) If M is of type $(A_1, \ldots, A_n)$ then sel[i] M is a term of type A_i for any integer i in $1, \ldots, n$. The term sel[i] M denotes the operation of selecting the ith component of M.

  These two term-formers satisfy the equations

$$\mathtt{sel}[i](M_1, \ldots, M_n) \equiv M_i$$
$$(\mathtt{sel}[1]M, \ldots, \mathtt{sel}[n]M) \equiv M$$

  The first equation says that building a tuple $(M_1, \ldots, M_n)$ and then unbuilding it by a selection operator for the ith component has the same effect as M_i. The second equation says that unbuilding a tuple M and rebuilding it has no net effect.
- Whenever A and B are types, there is a type $A \to B$ that we think of as the type of "functions" from A to B. In ordinary usage, we define functions by writing equations that specify their action on prototypical inputs, for example, $f(x) = M$. We are really saying here that f is "the function that maps x to the corresponding value of M." From a type-theoretic point of view, it is better to introduce a term-former that denotes this construction. The notation $\lambda x.M$ is used to denote the function that maps x to the corresponding value of M. So,

  (1) If M is a term of type B that (possibly) uses a free variable x of type A, then $\lambda x.M$ is a term of type $A \to B$.
  (2) If M and N are terms of type $A \to B$ and A, respectively, then $M\ N$ is a term of type B. This denotes the operation of applying the function value of M to the value of N. The notation $M(N)$ is also used, but the clutter of the parentheses is really unnecessary.

  The variable x is said to be bound in the term $\lambda x.M$. To formalize the variable binding features as well as the type correctness conditions, it is conventional to give *type rules* for the term-formers. These are shown in Fig. 5 for both the product and function type constructors. The symbol Γ stands for a finite collection of typings for distinct variables such as $x_1{:}A_1, \ldots, x_n{:}A_n$. The statement Γ ($M{:}A$ means that "the term M has the type A assuming that its free variables have the types listed in Γ." The fact that λ binds a variable is

$$\frac{\Gamma \rhd M_1 : A_1 \cdots \Gamma \rhd M_n : A_n}{\Gamma \rhd (M_1, \ldots, M_n) : (A_1, \ldots, A_n)}$$

$$\frac{\Gamma \rhd M : (A_1, \ldots, A_n)}{\Gamma \rhd \mathtt{sel}[i]\, M : A_i}$$

$$\frac{\Gamma, x{:}A \rhd M : B}{\Gamma \rhd \lambda x.M : A \to B}$$

$$\frac{\Gamma \rhd M : A \to B \qquad \Gamma \rhd N : A}{\Gamma \rhd M N : B}$$

Fig. 5. Type rules for product and function type constructors.

represented by deleting this variable from the free-variable list in the consequent of the type rule. The equations for the term-formers are

$$\begin{aligned}(\lambda x.M)N &\equiv M[N/x]\\ \lambda x.Mx &\equiv M \qquad \text{if } x \text{ is not free in } M\end{aligned}$$

The first equation states the effect of building a function and then "unbuilding" it by application to an argument. The net effect is to use N in place of x in the term for the function. The second equation says that the function that maps x to $M(x)$ is the same as M.

The equations that underlie type theory are fundamental. They have to do with the inherent meaning of the data structures or computational structures such as tupling and functions. While they are best known in the context of functional programming, their applicability is not limited to functional programming. They apply wherever type constructors of tupling and function spaces are involved. We will see in the section entitled "Procedures and Objects" their application in the context of imperative and object-oriented programming.

Textbooks on semantics (1,7,8), have a detailed treatment of type theories. It has been found that more sophisticated type theories can be used for encoding properties of programs as well as proving them (9,10). These theories exploit a certain correspondence between types and propositions in intuitionistic logic called *Curry–Howard correspondence*. Category theory provides a more mathematical (and abstract) treatment of type theory with wide-ranging applications. Texts (see Refs. 11 and 12) have a detailed treatment, while Ref. (13) is a gentle introduction to the subject.

General Recursion. In the examples of this section (sort and partition), we have taken care to write recursive programs so that they denote well-defined functions. This is done by ensuring that the recursive calls are made to smaller arguments. However, usual programming languages allow *unrestricted* recursion. The functions denoted by programs may then be partial functions that are undefined for some inputs and defined for others. In computational terms, such undefinedness gets exhibited as nontermination. For example, the recursive definition

```
f:Int → Int
f n = if n = 0 then 1
else n ∗ (n − 1) ∗ f(n − 2)
```

defines a partial function: f maps any non-negative even integer n to the factorial of n, it is undefined for the other integers.

The general type-theoretic situation is as follows. If $F{:}t \to t$ is a function, there is a value (rec F of type t that satisfies

$$\mathtt{rec}\, F = F\ (\mathtt{rec}\, F)$$

We then express the about function f as:

$$\begin{aligned}&\mathtt{rec}\ \lambda \mathtt{f}.\ \lambda \mathtt{n}.\ \mathtt{if}\ \mathtt{n} = 0\ \mathtt{then}\ 1\\ &\qquad\qquad \mathtt{else}\ \mathtt{n} * (\mathtt{n}-1) * \mathtt{f}(\mathtt{n}-2)\end{aligned}$$

The value (rec F) is called a *fixed point* of F because it remains unchanged under the action of F.

To deal with recursion in general, we need a theory of *partial elements*. Such a theory was developed by Scott (14), based on the classical work of Kleene in recursive function theory (15).

We consider sets D together with a specified partial order $\sqsubseteq_D$. The partial orders are used to model definedness: $x \sqsubseteq_D y$ means that x is "less defined" than or equal to y. For example, the set of partial functions $[A \rightharpoonup B]$ between sets A and B can be partially ordered by defining that $f \sqsubseteq g$ iff, whenever $f(x)$ is defined, $g(x)$ is defined and equal to $f(x)$. A partially ordered set $\langle D, \sqsubseteq_D\rangle$ is called a *complete partial order* (or *cpo*, for short) if

- there is a *least* element $\perp_D \in D$ such that $\perp_D \sqsubseteq_D x$ for all $x \in D$, and
- whenever $x_0 \sqsubseteq_D x_1 \sqsubseteq_D x_2 \sqsubseteq_D \cdots$ is an increasing sequence (possibly infinite), there is an element $x^\infty \in D$ that the least upper bound of the sequence, that is, (1) x^∞ is greater than or equal to every x_i, and (2) if z is greater than or equal to every x_i, then $x^\infty \sqsubseteq_D z$.

The idea is that the least upper bound x^∞ captures the information of all the approximations x_i and nothing more. It can be verified that $[A \rightharpoonup B]$ forms a cpo. A function $F{:}D \to E$ between cpo's is said to be *continuous* if it preserves the least upper bounds of increasing sequences. All the functions definable in usual programming languages are continuous.

If $a = F(a)$ is a recursive definition of a value $a \in D$, where $F{:}D \to D$ is a continuous function, then the interpretation is that a is the least value such that $a = F(a)$ holds. Such a value is called the *least fixed point* of F. It is a result of Kleene that the least fixed point always exists: it is the least upper bound of the sequence

$$\perp_D \sqsubseteq_D F(\perp_D) \sqsubseteq_D F^2(\perp_D) \sqsubseteq_D \cdots$$

To prove properties of recursively defined values, one uses the fixed-point induction principle. Let $P(x)$ be a property for values $x \in D$ that includes the least upper bounds of increasing sequences, that is, whenever $x_0 \sqsubseteq_D x_1 \sqsubseteq_D \cdots$ is an increasing sequence such that $P(x_i)$ holds for each x_i, then $P(x^\infty)$ holds for the least upper bound x^∞. Such a property P is called an *inclusive predicate*. To prove $P(a)$ for a recursively defined value $a = F(a)$, it is enough to show

(1) $P(\perp_D)$, and
(2) $P(x) \to P(F(x))$ for all $x \in D$.

We show an example. Consider proving that $\mathrm{f} \sqsubseteq \lambda \mathrm{n}.\mathrm{n}!$ where f is the recursively defined partial function given before and n! is the factorial of n (undefined if n is negative). In other words, we are showing that, whenever f(n) is defined, its value is the factorial of n. We first verify that the property $\mathrm{P(f)} \iff \mathrm{f} \sqsubseteq \lambda \mathrm{n}.\mathrm{n}!$ is inclusive. The two conditions for the fixed-point induction are verified as follows:

(1) $\bot \sqsubseteq \lambda n.n!$. This is immediate from the fact that $\bot$ is the least element.

(2) $f \sqsubseteq \lambda n.n! \rightarrow (\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (n-1) * f(n-2)) \sqsubseteq \lambda n.n!$. By assumption, whenever $f(n-2)$ is defined, it is equal to $(n-2)!$. So, whenever $n * (n-1) * f(n-2)$ is defined, it is equal to $n!$. Moreover, $1 = 0!$. Thus, we have the conclusion.

Textbooks on semantics (1,7,8) as well as those on program verification (16,17) have a detailed discussion of fixed-point theory and proof methods for recursively defined functions. The introduction of partially defined elements has implications for the type theory. In particular, the theory for call-by-name programming languages and call-by-value languages diverges. The texts cited on semantics contain discussion of the differences.

Abstract Data Types

An abstract data type (*ADT*) is an implementation of a data structure via a collection of specified operations. The client programs that use the ADT can manipulate the values of the abstract type only by using the provided operations. They do not have direct access to the data representation used for implementing the type. A variety of programming languages such as Modula-2, Ada, and Standard ML include module facilities for defining ADTs. Specifying the behavior of such ADTs and ensuring that implementations meet the specifications are important concerns for a software engineer.

Two kinds of methods are followed for the correctness of ADTs:

- Axiomatic or Algebraic Method The behavior of the ADT is specified by a collection of axioms. Any implementation that satisfies the axioms is deemed correct.
- Equivalence method The behavior is specified by giving a naive implementation for the ADT without concern for efficiency. The correctness of any other implementation is proved by showing that it is equivalent to the naive implementation.

Note that these two methods parallel the two methods we have seen for the correctness of functions (an independent specification for quick sort, and a naive program for reverse). We illustrate the two methods for the data structure of queues.

Axiomatic Specifications. Figure 6 shows an axiomatic specification for queues of integers. The specification consists of three parts: First, the type of the data structure being specified is given (Queue). Second, the operations on the data structure are listed (empty, insert, . . .). Third, a collection of equational axioms for the operations are specified. Note that no definitions are given for the type Queue or the operations. An implementation of the ADT is free to choose definitions for them in such a way that the axioms are satisfied.

In understanding the specification, keep in mind that the operations are genuine functions with no "side effects." For example, the insert operation, given an element x and a queue data structure q, returns a *new* queue data structure that contains all the elements of q and the additional element x. How to define insert without excessive copying is a matter addressed in the implementation. Recall that queues are first-in–first-out data structures. So, insertions are done at the tail end of the structure and deletions at the front. The first three axioms capture this behavior. The first axiom is trivial while the second says that deleting the front of a singleton queue gives the empty queue. The third axiom says that inserting x at the end of a nonempty queue and then deleting the front has the same effect as doing these operations in the opposite order. The remaining axioms can be understood in a similar fashion.

```
type Queue
operations
    empty  : Queue
    insert : (Int, Queue) → Queue
    delete : Queue → Queue
    isempty: Queue → Bool
    front  : Queue → Int
axioms
    (1) delete(empty) = empty
    (2) delete(insert(x, empty)) = empty
    (3) delete(insert(x, insert(y, q)) = insert(x, delete(insert(y, q)))
    (4) isempty(empty) = true
    (5) isempty(insert(x, q)) = false
    (6) front(insert(x, empty)) = x
    (7) front(insert(x, insert(y, q))) = front(insert(y, q))
```

Fig. 6. Axiomatic specification for queues.

One might wonder if the third axiom could be written more simply as

$$\begin{aligned}&q \neq \text{empty} \Rightarrow\\ &\text{delete}(\text{insert}(x, q)) = \text{insert}(x, \text{delete}(q))\end{aligned}$$

Unfortunately, this statement is not quite acceptable because it uses the inequality predicate and we have not given any axioms for inequality. But the following restatement is meaningful:

$$\begin{aligned}&\text{isempty}(q) = \text{false} \Rightarrow\\ &\text{delete}(\text{insert}(x, q)) = \text{insert}(x, \text{delete}(q)).\end{aligned}$$

A natural question that arises is whether the specification is "correct" and even what it would mean for it to be "correct." Two criteria are often used:

- Consistency An ADT specification is consistent if it does not equate any two distinct values of predefined types (types other than the one being specified). The consistency criterion ensures that the axioms are reasonable (even though they might still be "wrong" in the sense that they might not capture the intended behavior). For example, if we replace axiom (7) by the following:

  (7′) front(insert(x, q)) = front(q)

  then it follows that any two values of type Int are equal:

  $x =_{(6)} \text{front}(\text{insert}(x, \text{empty})) =_{(7')} \text{front}(\text{empty}) =_{(7')} \text{front}(\text{insert}(y, \text{empty})) =_{(6)} y$

  The axiom (7′) is thus wrong because it leads to an inconsistency.
- Sufficient Completeness An ADT specification is sufficiently complete if it equates every term of a predefined type to some value of that type. This criterion ensures that we have enough axioms in the specification. For example, if we delete the axiom (4), then the term isempty(empty) is not equal to any value of type Bool.

Note that the specification of Fig. 6 is not in fact sufficiently complete because the term front(empty) is not equated to any value of type Int. Intuitively, front(empty) should be undefined because an empty queue does not have a front element. If we are interpreting types as cpo's rather than sets, we can use the axiom

```
type Queue = [Int]

empty = []
insert(x, q) = q ++ [x]
delete []= []
delete(x:q) = q
isempty []= true
isempty(x:q) = false
front(x:q) = x
```

Fig. 7. A list implementation of queues.

```
delete(empty) = delete []= []= empty
delete(insert(x, empty)) = delete [x] = []= empty
delete(insert(x, insert(y, q))) = delete(q ++ [y] ++ [x])
     = delete(q ++ [y]) ++ [x]
     = insert(x, delete(insert(y, q)))
```

Fig. 8. Verification of the list implementation.

front(empty) = ⊥

For set-theoretic types, the notion of "error values" has been proposed (18) to solve this problem.

These concerns indicate that writing axiomatic specifications is a rather delicate task. Considerable mathematical maturity is required to develop trustworthy specifications. A vast body of theory has been developed for facilitating this task (see Refs. 19, 20 and 21).

Models. Recall that an axiomatic specification introduces a type name (the abstract type) and a collection of operation names of specified types. These two pieces of data form what is called the *signature* of the abstract type. By picking a specific type to serve as the representation for the abstract type and specific functions to serve as the implementation of the operations, we obtain what is called a *structure*. A structure that satisfies the axioms of the specification is called a *model*. (The term *algebra* is also used to refer to models in our sense.) One way to implement abstract types is by giving models.

Figure 7 shows an implementation of queues using the representation of lists. The elements of a queue are stored in a list in the order in which they are to be deleted. Hence, insert is defined to add an element at the end of the list. The operations delete and front are implemented by the tail and head operations on lists, respectively.

To verify that the implementation forms a model, one merely proves the axioms in the specification for the particular functions defined in the implementation. For example, we show, in Fig. 8, the verification of the first three axioms for the list implementation of queues. Note that simple equational reasoning suffices. For the third axiom, we rely on the following lemma, which can be proved by induction on the length of q.

Lemma 5. *For all lists q and q′ such that* $q \neq []$, *delete*$(q ++ q') =$ *delete*$(q) ++ q'$.

Equivalence. The equivalence method for ADT implementations eschews the idea of specifications. We prove the correctness of an implementation by showing that it is equivalent to a naive implementation whose correctness is taken to be obvious. The central issue in such a proof is to recognize that the two implementations might use quite different representations for the abstract type. So, it is not possible to talk about the *equality* of representations in the two implementations.

```
type Queue = ([Int], [Int])

empty = ([], [])
insert(x, (f, r)) = reform(f, x:r)
delete([], r) = ([], r)
delete(x:f, r) = reform(f, r)
isempty([], r) = true
isempty(x:f, r) = false
front(x:f, r) = x

reform([], r) = (rev(r), [])
reform(x:f, r) = (x:f, r)
```

Fig. 9. Melville implementation of queues.

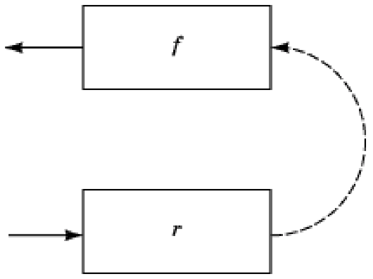


Fig. 10. Diagrammatic view of Melville queues.

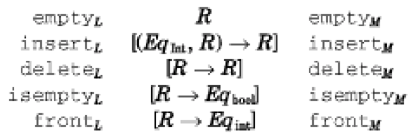
The solution, developed by Milner (22), Hoare (23), and Reynolds (24), is to use a binary relation called *simulation* between the two representation types. Suppose X and Y are the two representation types. We choose a relation $R{:}X \leftrightarrow Y$, which relates $x \in X$ and $y \in Y$ precisely when they have identical effect in the two implementations. The equivalence of the representations can then be shown by using the relation R in place of equality.

To make these ideas concrete, consider the Melville implementation of queues (25) shown in Fig. 9. The elements of the queue are split into two lists f and r. The queue elements in f are stored in the order they are to be deleted and those in r are stored in the order they are inserted. See Fig. 10. When the f part of the representation becomes empty, we reverse the r part and store it in f. This is done by the function reform. (Since list reversal can be done in linear time, this gives a constant-time amortized cost for the queue operations.) Furthermore, we arrange matters so that the f part of the representation is nonempty whenever the r part is nonempty.

Now, consider proving that the Melville implementation is equivalent to the list implementation of Fig. 7. We need to define a relation R:[Int] $\leftrightarrow$ ([Int], [Int]) such that it relates the representations that have equivalent effect in the two implementations. The following definition serves the purpose:

$$\begin{aligned} \mathrm{q}\,\boldsymbol{R}\,(\mathrm{f},\mathrm{r}) &\iff (\mathrm{q} = \mathrm{f} \mathbin{+\!\!+} (\mathrm{rev}\,\mathrm{r})) \wedge \boldsymbol{I}(\mathrm{f},\mathrm{r}) \\ \textbf{where } \boldsymbol{I}(\mathrm{f},\mathrm{r}) &\iff (\mathrm{f} = [] \implies \mathrm{r} = []) \qquad \textbf{(1)} \end{aligned}$$

The relation treats a list q and a Melville representation (f, r) as equivalent if



$\texttt{empty}_L$	R	$\texttt{empty}_M$
$\texttt{insert}_L$	$[(Eq_{\mathrm{Int}}, R) \to R]$	$\texttt{insert}_M$
$\texttt{delete}_L$	$[R \to R]$	$\texttt{delete}_M$
$\texttt{isempty}_L$	$[R \to Eq_{\mathrm{bool}}]$	$\texttt{isempty}_M$
$\texttt{front}_L$	$[R \to Eq_{\mathrm{int}}]$	$\texttt{front}_M$

Fig. 11. Verification conditions for queue operations.

(1) q consists of the elements of f followed by the elements of r in the reverse order, and
(2) f is empty only if r is empty.

Note that the second condition is independent of q. It is a condition that all good Melville representations must satisfy. Such conditions are often called *representation invariants*.

To formulate the verification conditions for the queue operations, we first introduce some notation for relations.

- For any type A, Eq_A:$A \leftrightarrow A$ is the equality relation.
- If R_1:$A_1 \leftrightarrow A'_1, \ldots, R_n$:$A_n \leftrightarrow A'_n$ are relations, then there is a relation $(R_1, \ldots, R_n)$:$(A_1, \ldots, A_n) \leftrightarrow (A'_1, \ldots, A'_n)$ between the tuple types, defined by

$$(x_1, \ldots, x_n)(R_1, \ldots, R_n)(y_1, \ldots, y_n) \iff \forall i = 1, \ldots, n\, x_i R_i y_i$$

- If R:$A \leftrightarrow A'$ and S:$B \leftrightarrow B'$ are relations, then there is a relation $[R \to S]$:$[A \to B] \leftrightarrow [A' \to B']$ between the function spaces, defined by

$$f[R \to S]f' \iff (\forall x, x' . xRx' \implies f(x)Sf'(x'))$$

Using these notations, for every type expression $F(a)$ over an abstract type a, we can define a parallel relational expression $F(R)$ that extends a relation R:$X \leftrightarrow Y$ to a relation $F(R)$:$F(X) \leftrightarrow F(Y)$. The definition is as follows:

- If $F(a) = a$, then $F(R) = R$.
- If $F(a) = A$, where A is a type other than a, then $F(R) = Eq_A$.
- If $F(a) = (F_1(a), \ldots, F_n(a))$, then $F(R) = (F_1(R), \ldots, F_n(R))$.
- If $F(a) = [F_1(a) \to F_2(a)]$, then $F(R) = [F_1(R) \to F_2(R)]$.

The relations $F(R)$ defined in this fashion are called *logical relations* and they have a long history in type theory (1). A *simulation relation* between two ADT implementations is a relation R between their representation types such that all the corresponding operations satisfy the logical relation $F(R)$. If there is a simulation relation between two ADT implementations then the implementations are equivalent.

Figure 11 lists the verification conditions for showing that the relation R defined in Eq. (1) is a simulation relation. We are using the subscripts L and M for the operations in the list implementation and Melville implementation, respectively.

The verification conditions are easy to check. As a sample, we prove the condition for delete. First, note that the reform function satisfies the property

reform(f, r) = (f′, r′) ⇒
(f ++ (rev r) = f′ ++ (rev r′)) ∧ I(f′, r′)

In other words, the reform function establishes the representation invariant without altering the contents of the queue.

Now, the verification condition for delete can be expanded as

(q = f ++ (rev r) ∧ I(f, r)) ⇒
delete_L(q) R delete_M(f, r)

Consider two cases:

- *Case* f = []. We have r = [] by I(f, r), and hence q = []. So, delete_L(q) = [] and delete_M(f, r) = ([], []). These are related by R.
- *Case* f = x:f′. We have q = x:f′ ++ (rev r). So, delete_L(q) = f′ ++ (rev r) and delete_M(f, r) = reform(f′, r). These are related by R.

The verification conditions for the other operations can be checked similarly.

Why does this method work? Intuitively, the verification conditions ensure that (1) whenever the same sequence of operations is carried out to build a queue, then the queues obtained in the two implementations are related by the simulation relation, and (2) whenever the same sequence of operations is carried out to observe queues related by the simulation relation, we obtain identical observable values. Thus, the existence of any simulation relation between the two implementations implies that the implementations are behaviorally equivalent.

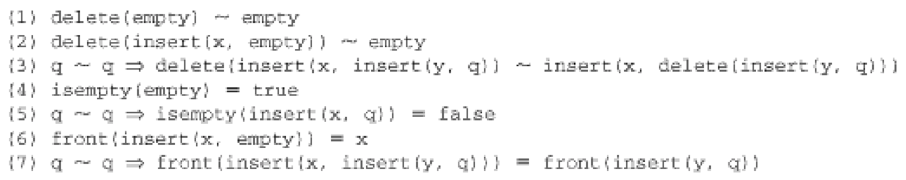
A variant of the simulation method, popularized by Hoare (23), relies on the fact that the simulation relation is often a partial function from the more concrete representation type to the more abstract representation type. For example, we can define the correspondence between Melville queues and list representations of queues by the function

abs: ([Int], [Int]) → [Int]
abs(f, r) = f ++ (rev r)

The Melville representation is more concrete in the sense that it has multiple representations that correspond to the same abstract queue. (The queue elements can be split between the f and r parts in different ways.) The same reasoning as before shows that all the queue operations preserve the abs function. Structure-preserving functions of this kind are called *homomorphisms* and have a long history in mathematics. The majority of the literature on abstract types (21) uses homomorphisms to relate different data representations. All these ideas, in fact, work more generally for simulation relations. The relational method is discussed, with numerous examples, in Refs. (26) and 27, but they treat imperative programs. Some of the research articles discussing the relational method include Refs. 28 to 30.

Implementations and Models. In the section entitled "Models," we have seen that models of axiomatic specifications provide valid implementations. However, implementations might also be behaviorally equivalent to models even if they are not models themselves. Such implementations are certainly acceptable. For example, the Melville implementation of queues does not form a model of the axiomatic specification of queues. [For instance, the axiom (3) does not hold if q = ([0] [1])]. However, it is behaviorally equivalent to the list implementation which is a model.

There is a general technique for identifying the model underlying an implementation (if there is one) (31). An implementation may fail to be a model for two kinds of reasons: (1) some of the values of the representation



```
(1) delete(empty) ~ empty
(2) delete(insert(x, empty)) ~ empty
(3) q ~ q ⇒ delete(insert(x, insert(y, q)) ~ insert(x, delete(insert(y, q)))
(4) isempty(empty) = true
(5) q ~ q ⇒ isempty(insert(x, q)) = false
(6) front(insert(x, empty)) = x
(7) q ~ q ⇒ front(insert(x, insert(y, q))) = front(insert(y, q))
```

Fig. 12. Verification conditions for implementations of queues.

type may be unused, and (2) multiple values of the representation type may represent the same abstract value. By factoring out these differences, we can obtain a model.

The formalization of the idea is as follows. A *partial equivalence relation* (*PER*) is a binary relation $\sim : X \leftrightarrow X$ that is symmetric and transitive (but not necessarily reflexive). The subset $\{x \in X \mid x \sim x\}$ is called the *domain* of $\sim$, and denoted dom($\sim$). Note that the relation $\sim$ reduces to an ordinary equivalence relation over dom($\sim$). Typically, we use a representation invariant to identify dom($\sim$). For every $x \in \mathrm{dom}(\sim)$, there is an $\sim$-*equivalence class*, which is the set of all values equivalent to x, denoted $[x]$. Note that $[x] = [y]$ if and only if $x \sim y$. The set of all $\sim$-equivalence classes is denoted $X/\sim$.

For the simple type expressions that we are considering, it turns out that, if a PER $\sim : X \leftrightarrow X$ is a simulation relation between an implementation and itself then there is a behaviorally equivalent implementation using $X/\sim$ as the representation type. We only need to ensure that this derived implementation is a model for the original one to be a valid implementation. Moreover, since the equality relation of $X/\sim$ corresponds to the relation $\sim$, it is possible to formulate verification conditions for the implementation without mentioning the derived implementation explicity.

To illustrate this, we show in Fig. 12, the verification conditions for showing that an implementation equipped with a PER $\sim$ meets the axiomatic specification of queues. Note that (1) every free variable of type Queue is restricted to lie within the domain of the relation $\sim$, and (2) the equality relation for queues is replaced by $\sim$. These changes reflect the fact that it is the derived implementation over equivalence classes that is being verified to be a model.

The correctness of the Melville implementation of queues can be verified using the following PER:

$$
\begin{array}{l}
(\mathrm{f}, \mathrm{r}) \sim (\mathrm{f}', \mathrm{r}') \iff I(\mathrm{f}, \mathrm{r}) \wedge \\
I(\mathrm{f}', \mathrm{r}') \wedge (\mathrm{f} ++ (\mathrm{rev\ r}) = \\
\mathrm{f}' ++ (\mathrm{rev\ r}'))
\end{array}
$$

The relation treats two representations as being equivalent if they have the same queue elements (assuming they are valid representations satisfying the invariant).

Imperative Programs

In this section, we review correctness methods for an entirely different programming model, viz., that of basic imperative programs. In this model, we consider mutable variables, assignment commands and control structures. Procedures and other high-level mechanisms are postponed to the next section. Denotational methods as for functional programs are still applicable to this programming model. However, it will be seen that axiomatic methods are somewhat more effective owing to the specialized nature of commands.

The concept of *variable* is central to imperative programming. A variable is an abstract *storage cell* that holds a specific value and this value can be altered during the execution of a program. It is important to distinguish this from the notion of variable encountered in functional programming. Variables there were *symbols* that stand for arbitrary values of some type. The variables of imperative programming are *not* symbols, though we often use symbols to name variables. Some languages and formalisms fuse the two notions of variables into one. We keep them separate. In this section and the next, symbols, that is, variables in the sense of functional programming, are called *identifiers*, and the term variable is reserved for storage cells.

A basic imperative program is written over a fixed collection of variables, designated by separate identifiers. The types of these variables are specified via declarations such as

```
var x,y:Int
```

Types such as Int are called *data types*. Variables and expressions can take values of data types. A program is a *command* that is made up of

- assignments of the form $X := E$ where X is a variable and E an expression,
- the trivial command skip,
- sequencing operation $C_1; C_2$,
- conditional construction if B then C_1 else C_2, where B is a boolean expression, and
- loops of the form while B do C, where B is a boolean expression.

The structure of expressions is standard; it is made of variable identifiers, constants, and the usual operations appropriate for various data types. In an assignment command $X := E$, the variable X and the expression E are required to be of the same type.

It is conventional to treat arrays as forming a data type. The values of an array type Array t are partial functions from integers to t, whose domain is a contiguous range of integers $i, \ldots, j$. The subscripting expression $a[p]$ produces the pth element of a, and $a[p \to x]$ denotes the modified partial function with the pth element mapped to x. Both the expressions are undefined if p is not a valid index into the array. If a is an array variable, the assignment command a[p] $:= E$ is regarded as a notational variant of a := a[p $\to E$].

The denotational model of the basic imperative language is defined using the idea of *states*. Given a declaration for a collection of variable identifiers $X_1, \ldots, X_n$, a state is a mapping $[X_1 \mapsto v_1, \ldots, X_n \mapsto v_n]$ such that each v_i is a value of the type of X_i. Let *State* denote the set of all such states. If $s \in State$, we write $s(X)$ for the value assigned to X in the state s, and $s[X \to v]$ for the state that is the same as s except that it maps X to v.

Expressions of type t are interpreted as partial functions $State \rightharpoonup t$. We call such functions *state valuations*. In particular, a variable X used as an expression denotes the function $\lambda s.s(X)$. An expression of the form $E_1 + E_2$ denotes the function $\lambda s.E_1(s) + E_2(s)$.

Commands are interpreted as *state transformations*, that is, partial functions of type $State \rightharpoonup State$.

- The *assignment* $X := E$ denotes the partial function $\lambda s.s[X \to E(s)]$.
- The trivial command skip denotes the identity transformation $\lambda s.s$.
- A *sequencing* command $C_1; C_2$ denotes the partial function $\lambda s.C_2(C_1(s))$.
- A *conditional* command if B then C_1 else C_2 denotes the partial function

$$\lambda s. \text{if } B(s) \text{ then } C_1(s) \text{ else } C_2(s)$$

- A *loop* command while B do C denotes the recursively defined partial function w defined by

$$w(s) = \text{if } B(s) \text{ then } w(C(s)) \text{ else } s$$

Using this denotational model, it is easy to verify a number of simple equivalences for commands:

$$\begin{aligned}
&C; \texttt{skip} \equiv C \equiv \texttt{skip}; C \\
&(C_1; C_2); C_3 \equiv C_1; (C_2; C_3) \\
&(X := E_1[X]; X := E_2[X]) \\
&\quad \equiv (X := E_2[E_1[X]]) \\
&\text{if } B \texttt{ then } C_1 \texttt{ else } C_2 \\
&\quad \equiv \texttt{ if not } (B) \texttt{ then } C_2 \texttt{ else } C_1 \\
&\texttt{while } B \text{ do } C \equiv \texttt{ if } B \texttt{ then } (C; \texttt{ while } B \text{ do } C) \texttt{ else skip}
\end{aligned}$$

The commutativity property $C_1; C_2 \equiv C_2; C_1$ does not hold in general because C_1 can affect variables that are used in C_2 or vice versa. However, there are important special cases in which such reordering is valid. For instance, if C_1 and C_2 do not share any free identifiers, one expects the reordering to be valid. We consider a more general situation.

Definition 6. *A free identifier X of a term T is called a passive free identifier of T if all its occurrences are within expressions. Otherwise, it is called an active free identifier. Two terms T_1 and T_2 are said to be noninterfering if all their common free identifiers are passive in both T_1 and T_2. We write this fact symbolically as T_1 # T_2.*

The idea is that the passive free identifiers of a term denote variables that are used in a "read-only" fashion. If two terms are noninterfering, none of them writes to any variables used in the other term. So, the execution or evaluation of one term does not affect the meaning of the other. For example, the two commands x := x + z and y := y ∗ z are noninterfering because their only common free identifier is z which is used passively in both terms.

Theorem 7. If C_1 and C_2 are noninterfering commands then $C_1; C_2 \equiv C_2; C_1$.

Since the denotational model of the basic imperative language is in terms of functions, one might expect that the standard reasoning techniques for functions are applicable to them. This is certainly the case for simple programs. For example, the following program exchanges the values of variables x and y using an auxiliary variable t for temporary storage:

$$C \equiv (\texttt{t} := \texttt{x}; \texttt{x} := \texttt{y}; \texttt{y} := \texttt{t})$$

It is easy to prove the correctness statement:

$$\text{For all states } s, C(s)(\texttt{x}) = s(\texttt{y}) \text{ and } C(s)(\texttt{y}) = s(\texttt{x})$$

by calculating $C(s) = s[\text{t} \to s(\text{x})][\text{x} \to s(\text{y})][\text{y} \to s(\text{x})]$. However, this kind of reasoning involves excessive manipulation of states. Since states are never explicity mentioned in imperative programs, it is preferable to devise logical notations that operate at a high-level without mentioning states. The notation of Hoare triples (32) is the most widely used notation for this purpose.

$$\frac{}{\{P[E/X]\}\,X := E\,\{P\}}\ \textbf{Assign}$$

$$\frac{}{\{P\}\,\mathtt{skip}\,\{P\}}\ \textbf{Skip}$$

$$\frac{\{P\}\,C_1\{Q\} \qquad \{Q\}\,C_2\{R\}}{\{P\}\,C_1;\,C_2\{R\}}\ \textbf{Sequencing}$$

$$\frac{\{P \wedge B\}\,C_1\{Q\} \qquad \{P \wedge \mathrm{not}(B)\}\,C_2\{Q\}}{\{P\}\,\mathtt{if}\,B\,\mathtt{then}\,C_1\,\mathtt{else}\,C_2\{Q\}}\ \textbf{Conditional}$$

$$\frac{\{P \wedge B\}\,C\,\{P\}}{\{P\}\,\mathtt{while}\,B\,\mathtt{do}\,C\,\{P \wedge \mathit{not}(B)\}}\ \textbf{While}$$

Fig. 13. Program rules of Hoare logic.

A Hoae triple is a formula written using the notation

$$\{P\}C\{Q\}$$

where P and Q are generalized boolean expressions called *assertions* and C is a command. The triple is a logical statement that means

$$\text{For all states } s \text{ and } s',$$
$$\text{if } P(s) = \text{true and } C(s) = s',\ \text{then } Q(s') = \text{true}$$

Informally, this says that, in any initial state in which P is true, if the execution of the command C terminates then the assertion Q is true in the final state. Note that nothing is said in case the execution of C does not terminate. For this reason, Hoare triples are called *partial correctness* statements. (It is also possible to devise a Hoare triple notation for total correctness, but rules for their manipulation are more involved.) The assertion P is called the precondition or the input assertion and Q the post-condition or output assertion.

An example of a valid Hoare triple is

$$\{\mathrm{x} \geq 0\}\mathrm{x} := \mathrm{x} + 1\{\mathrm{x} > 0\}$$

In any state in which x is non-negative, incrementing x leads to a state in which x is positive. The correctness of the variable-swapping command C can be formulated by the statement

$$\text{for all } a, b : t,\ \{\mathrm{x} = a \wedge \mathrm{y} = b\}\,C\,\{\mathrm{x} = b \wedge \mathrm{y} = a\}$$

Here, we have used two value identifiers a and b to record the initial values of x and y. They are not variables and so, cannot be modified. Such identifiers are sometimes called logical variables. In our terminology, they are not variables but identifiers.

Valid Hoare triples can be inferred using a system of if–then rules without ever mentioning explicit states. This system of rules is called *Hoare logic* and shown in Figs. 13 and 14. In addition to Hoare triples, the logic uses a logical statement of the form {P}, with the meaning that the assertion P is true in all states.

The rules of Fig. 13 deal with the various command forms. The Assign rule is somewhat surprising at first sight: an assertion P is true at the end of the assignment $X := E$ if the assertion $P[E/X]$, obtained by

$$\frac{\{P' \Rightarrow P\} \quad \{P\}\,C\,\{Q\} \quad \{Q \Rightarrow Q'\}}{\{P'\}\,C\,\{Q'\}}\ \text{Consequence}$$

$$\frac{\{P_1\}\,C\,\{Q_1\} \quad \{P_2\}\,C\,\{Q_2\}}{\{P_1 \wedge P_2\}\,C\,\{Q_1 \wedge Q_2\}}\ \text{Conjunction}$$

$$\frac{\{P_1\}\,C\,\{Q_1\} \quad \{P_2\}\,C\,\{Q_2\}}{\{P_1 \vee P_2\}\,C\,\{Q_1 \vee Q_2\}}\ \text{Disjunction}$$

$$\frac{P \# C}{\{P\}\,C\,\{P\}}\ \text{Constancy}$$

$$\begin{array}{c} \{P\} \\ \vdots \\ \dfrac{P \# C \quad \{Q\}\,C\,\{R\}}{\{P \wedge Q\}\,C\,\{P \wedge R\}} \end{array}\ \text{Strong Constancy}$$

Fig. 14. "Logical" rules for Hoare triples.

substituting E for all occurrences of X in P, is true before the assignment. What is surprising is that the substitution is working backwards. The post-condition determines the precondition, not the other way around. However, the forward-reasoning Hoare triple

$$\{\text{true}\}X := E\{X = E\}$$

is an instance of the Assign rule *provided X does not occur in E*. In that case, $(X = E)[E/X] \equiv (E = E) \equiv$ true. The Assign rule works even when X occurs in E. For example, the Hoare triple

$$\{\mathrm{x} \geq 0\}\mathrm{x} := \mathrm{x} + 1\{\mathrm{x} > 0\}$$

follows from the Assign rule because $(\mathrm{x} > 0)[\mathrm{x} + 1/\mathrm{x}] \equiv \mathrm{x} + 1 > 0$, which is equivalent to $\mathrm{x} \geq 0$. Why is the Assign rule sound? Suppose s is a state such that $P[E/X](s) = \text{true}$. A little thought reveals $P[E/X](s) = P(s[X \rightarrow E(s)])$. But $s[X \rightarrow E(s)]$ is nothing but $(X := E)(s)$, the final state of the assignment. Hence, P holds in the final state.

The rules Skip, Sequencing, and Conditional are straightforward. The while rule introduces the idea of an *invariant* assertion. The premise of the rule requires that whenever P and the loop condition B are true, the execution of the loop body C leads to a state in which P is again true. We say that C leaves the assertion P invariant. It is then easy to see that the entire loop (while B do C) leaves the assertion P invariant. Note that there is no requirement that the loop terminates. This is reasonable because Hoare triples are partial correctness statements.

The rules of Fig. 14 are termed logical rules because they derive from the logical meaning of Hoare triples and are independent of the commands involved. Since the interpretation of $\{P\}C\{Q\}$ is that if P is true in some initial state of C, then Q is true in the corresponding final state of C, the assertion P plays the role of a premise and the assertion Q plays the role of a conclusion. Hence, it is valid to replace P by a *stronger* assertion P' and Q by a *weaker* assertion Q'. The Consequence rule formalizes this. The rules Conjunction and Disjunction allow one to combine Hoare triples.

If P is independent of C, then the value of P is constant throughout the execution of C. Hence, $\{P\}C\{P\}$. This gives the Constancy rule.

The Strong Constancy rule is a more powerful version of Constancy, invented by Reynolds (chapter 6 of Ref. 38). If a command C does not affect an assertion P then, whenever P is true in the start state, it will continue to be true throughout the execution of C. Therefore, in proving properties of C, we can assume that P is true for *all states*. (P may not be actually true for all states. But it will be true for all the states that arise during the execution of C.) If P # C, we say that P is a *general invariant* in C.

Proofs in Hoare logic are often presented as *proof outlines*. These are programs annotated with assertions at strategic places. In particular, the beginning and ending of the program are annotated with input and output assertions. Every while loop is annotated with {whileinv I} where I is an assertion (the invariant for the loop). A proof outline is *valid* if

(1) for every segment of the form $\{P\}C\{Q\}$ or $\{P\}C$ {whileinv Q} in the outline, $\{P\}C\{Q\}$ is a valid Hoare triple, and

(2) for every segment of the form {whileinv I} while B do C; C' $\{Q\}$, the following are valid Hoare triples:

$$\{I \wedge B\}\, C\, \{I\}$$

$$\{I \wedge \mathrm{not}\,(B)\}\, C'\, \{Q\}$$

(3) for every block of the form

$$\texttt{begin}\ \{\mathrm{geninv}\ I\}\ C\ \texttt{end}$$

the condition I # C must be true.

A proof of correctness consists of a proof outline together with a proof of its validity.

Figure 15 shows a program for partitioning an array together with a proof outline. We assume that SWAP(a, p, q) is some command that is equivalent to

$$a := a[p \to a[q],\, q \to a[p]]$$

The input assertion for the program is

$(0 \le \mathrm{i} \le \mathrm{j} \le 99) \wedge (\mathrm{a} = \mathrm{a}_0)$

which specifies that indices i and j are within the array bounds and names the initial values of a to be a_0. The task is to partition the array segment a[i . . . j] using a[i] as the pivot. The program partitions the segment into three subsegments a[i . . . (mid − 1)], a[mid], and (a[(mid + 1) . . . j]) such that all the elements in the first segment are less than or equal to a[mid] (small values) and those in the last segment are greater than a[mid] (large values). This suggests the post-condition

$0 \le \mathrm{i} \le \mathrm{j} \le 99 \wedge \mathrm{a} \approx \mathrm{a}_0 \wedge$
$\mathrm{i} \le \mathrm{mid} \le \mathrm{j} \wedge$
$\mathrm{a}[\mathrm{i} \ldots (\mathrm{mid}-1)] \le \mathrm{a}[\mathrm{mid}] \wedge$
$\mathrm{a}[(\mathrm{mid}+1) \ldots \mathrm{j}] > \mathrm{a}[\mathrm{mid}]$

Here, $\mathrm{a} \approx \mathrm{a}_0$ means that a and a_0 have the same collection of elements. The notation $\mathrm{a}[p \ldots q] \le x$ means, for all k such that $p \le k \le q$, $\mathrm{a}[k] \le x$. Since i and j are passive free identifiers, the condition $0 \le \mathrm{i} \le \mathrm{j} \le 99$ is a general invariant in the program. Thus, by using the Strong Constancy rule, we can assume that it holds

```
var a: Array Real(100);
var i, j: Int;
var mid: Int;
var l, h: Int
begin
  {geninv 0 ≤ i ≤ j ≤ 99}
  l := i+1; h:= j;
  {whileinv W :(i ≤ (l−1) ≤ h ≤ j) ∧ (a[(i+1) ... (l−1)] ≤ a[i]) ∧ (a[(h+1) ... j] > a[i])}
  while (l−1) < h dobegin
    if a[l] ≤ a[i] then
      l := l+1
    else if a[h] > a[i] then
      h := h−1
    else
      (SWAP(a, l, h); l := l+1; h := h−1);
  end;
  mid := l−1;
  SWAP(a, i, mid)
  {F : (i ≤ mid < j) ∧ (a[i...(mid−1)] ≤ a[mid]) ∧ (a[(mid+1) ... j] > a[mid]) }
end
```

Fig. 15. Proof outline for array partitioning.

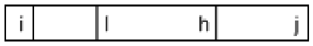


Fig. 16. The structure of array segments during partition.

in all states. The condition $a \approx a_0$ can be proved separately and is, in fact, rather obvious because the only changes made to a are via SWAP. (It is possible to regard this condition as a general invariant using a more sophisticated notion of general invariants. See Ref. 26.) The proof outline of Fig. 15 is meant for showing the remaining conditions of the output assertion.

The key to the proof of correctness is the invariant assertion for the while loop. In our solution, indices l and h are used to mark the low end and high end, respectively, of the array segment to be partitioned. Thus, the structure of the array segment is as shown in Fig. 16. (The notation used for this diagram is called *partition diagram*. It is formalized in 26.) The first subsegment contains the pivot element, the second and the fourth subsegments contain small and large values, respectively, and the middle subsegment contains unprocessed values. The invariant assertion is based on this analysis.

The task of proving correctness is now split into three parts (using W for the invariant, F for the final assertion):

(1) $\{0 \le i \le j \le 99\}$ l := i + 1; h := j $\{W\}$
(2) $\{W \wedge (l-1) < h\}$ loop-body $\{W\}$
(3) $\{W \wedge (l-1) \ge h\}$ mid := l − 1; SWAP(a, i, mid) $\{F\}$

For part 1, we see by assignment and sequencing rules that {true} l := i + 1; h := j $\{l = i + 1 \wedge h = j\}$. We need to show that the post-condition here implies W, which is a straightforward verification. Note that the segments $a[(i + 1) \ldots (j - 1)]$ and $a[(h + 1) \ldots j]$ are empty.

For part 2, we first check that l and h are proper subscripts for the array (since $i \le l - 1 < h \le j$). If $a[l] \le a[i]$, then $W \wedge (l-1) < h \rightarrow W[l + 1/l]$. If $a[h] > a[i]$, then $W \wedge (l-1) < h \rightarrow W[h - 1/h]$. Otherwise, l and h are distinct, and we verify $W \wedge (l-1) < l < h \rightarrow W[a'/a, (l + 1)/l, (h-1)/h]$. where $a' = a[l \rightarrow a[h], h \rightarrow a[l]]$.

For part 3, we verify $W \wedge (l-1) \ge h \rightarrow F[(l-1)/mid, a'/a]$ where $a' = a[i \rightarrow a[l-1], (l-1) \rightarrow a[i]]$.

This completes the proof of the partial correctness of the partitioning program. For termination, we note that $h - (l - 1) \geq 0$ is an invariant of the loop and the loop body monotonically decreases it, that is,

$$\text{for all } k : \texttt{Int}.\ k > 0 \Longrightarrow \\ \{h - (\mathtt{I} - 1) = k\}\ \text{loop body}\ \{h - (\mathtt{I} - 1) < k\}$$

Therefore, it follows by induction on k that, for all states s in which $h - (l - 1)$ has the value k, $w(s)$ is defined (where w is the state transformation function of the while loop).

Correctness proofs of this kind are fairly elementary to construct. See Manna (17), Loeckx and Sieber (16), or Mitchell (1) for a detailed treatment. The texts by Gries (33), Reynolds (26) and Jones (34) give an introductory treatment with numerous examples. These texts also describe techniques for program development with correctness proofs integrated into the process. A closely related system to Hoare logic is the weakest-precondition calculus of Dijkstra (35). A somewhat more structured framework for program development is the "refinement calculus" presented by Morgan (27,36).

Procedures and Objects

Procedures are *parametrized commands*. For example, the notation SWAP used in the partition program is a parameterized command, which can be defined as follows:

```
SWAP(a, p, q) ≡
begin var t: Real;
t := a[p]; a[p] := a[q]; a[q] := t
end
```

Mathematically, SWAP is a function that maps the parameter list (a, p, q) to a command. Thus the type theory of procedures is an instance of the type theory of functions provided we formalize the types of the parameters and results involved in imperative procedures. Such a type theory was first provided by Reynolds (37) and came to be called *Idealized Algol*.

The basic insight is to recognize that in addition to the data types that demarcate values storable in variables, there is another class of types called *phrase types*. Every class of phrases involved in the basic imperative language gives rise to a phrase type.

We use t to range over data types and θ to range over phrase types. The basic phrase types are

- Exp t for expressions that give t-typed values,
- Comm for commands, and
- Var t for variables that hold t-typed values.

In the logic for reasoning about programs, we also encounter the following phrase types:

- t for values of data type t, and
- Assert for assertions.

We adopt tuple types and function types from the type theory of functions:

- $(\theta_1, \ldots, \theta_n)$ for phrase types θ_i
- $\theta \to \theta'$ for phrase types θ and θ'

$$\text{Recordtypes}\quad \frac{\Gamma \triangleright \boldsymbol{M}_1 : \theta_1 \cdots \Gamma \triangleright \boldsymbol{M}_n : \theta_n}{\Gamma \triangleright [\mathrm{l}_1 = \boldsymbol{M}_1, \ldots, \mathrm{l}_n = \boldsymbol{M}_n] : [l_1 : \theta_1, \ldots, \mathrm{l}_n : \theta_n]}$$

$$\frac{\Gamma \triangleright \boldsymbol{M} : [\mathrm{l}_1 : \theta_1, \ldots, l_n : \theta_n]}{\Gamma \triangleright \boldsymbol{M}.l_i : \theta_i}$$

$$[l_1 = \boldsymbol{M}_1, \ldots, l_n = \boldsymbol{M}_n].l_i = \boldsymbol{M}_i$$
$$[l_1 = \boldsymbol{M}.l_1, \ldots, l_n = \boldsymbol{M}.l_n] = \boldsymbol{M}$$

$$\text{Class types}\quad \frac{\Gamma \triangleright \boldsymbol{K} : \mathtt{Cls}\ \theta' \qquad \Gamma, \boldsymbol{x} : \theta' \triangleright \boldsymbol{M} : \theta \qquad \Gamma, \boldsymbol{x} : \theta' \triangleright \boldsymbol{I} : \mathtt{Comm}}{\Gamma \triangleright (\mathtt{class\ new}\,\boldsymbol{K}\boldsymbol{x};\ \mathtt{methods}\ \boldsymbol{M};\ \mathtt{init}\ \boldsymbol{I}) : \mathtt{Cls}\ \theta}$$

$$\frac{\Gamma \triangleright \boldsymbol{K} : \mathtt{Cls}\ \theta \qquad \Gamma, \boldsymbol{x} : \theta \triangleright \boldsymbol{C} : \mathtt{Comm}}{\Gamma \triangleright \mathtt{begin\ new}\,\boldsymbol{K}\boldsymbol{x};\ \boldsymbol{C}\ \mathtt{end} : \mathtt{Comm}}$$

$$\mathtt{begin\ new}\ (\mathtt{class\ new}\,\boldsymbol{K}\boldsymbol{z};\ \mathtt{methods}\ \mathbf{M};\ \mathtt{init}\ \boldsymbol{I})\,\boldsymbol{x};\ \boldsymbol{C}\ \mathtt{end} \equiv \mathtt{begin\ new}\,\boldsymbol{K}\boldsymbol{z};\ \boldsymbol{I};\ \boldsymbol{C}[\boldsymbol{M}/\boldsymbol{x}]\ \mathtt{end}$$
$$(\mathtt{class\ new}\,\boldsymbol{K}\boldsymbol{x};\ \mathtt{methods}\ \boldsymbol{x};\ \mathtt{init\ skip}) \equiv \boldsymbol{K}$$

Fig. 17. Type theory of record and class types.

The reader would have noted that the type system of Idealized Algol differs considerably from the type systems of practical programming languages. For example, a type like Exp Int is rarely found in a typical programming. The point of Idealized Algol is to provide a clean and straightforward formal system to facilitate reasoning.

Using these phrase types, SWAP can be given the type

SWAP:(Var Array Real, Exp Int, Exp Int) $\rightarrow$ Comm

In general, functions with result type Comm correspond to procedures. Functions with result type Exp t correspond to function procedures or parametrized expressions. The type theory also allows functions whose results are variables, assertions, or other kinds of functions. See the papers in Ref. (38) for examples, especially Chaps. 8, 9, and 19.

For dealing with objects, we add two more phrase type forms:

- $[l_1{:}\theta_1, \ldots, l_n{:}\theta_n]$ is the type of *records* that have fields named $l_1, \ldots, l_n$ of respective types
- Cls θ is the type of *classes* that describe θ-typed objects

The type theory of these types is shown in Fig. 17. Here is a gentler explanation.

Record types are essentially notational variants of tuple types that allow the use of mnemonic field names for the components. Instead of writing a tuple as $(M_1, \ldots, M_n)$, we can write a record construction $[l_1 = M_1, \ldots, l_n = M_n]$, which builds a tuple and associates the field names $l_1, \ldots, l_n$ with the components. To select a field of a record R, we write $R.l_i$ instead of sel$[i]$ R. The record type $[l_1{:}\theta_1, \ldots, l_n{:}\theta_n]$ is thus isomorphic to the tuple type $(\theta_1, \ldots, \theta_n)$ and its operations satisfy laws similar to those of tuple types.

Objects are entities with hidden internal state and an externally accessible method suite. The methods are values of types that we have already seen: (possibly) parametrized commands and expressions, which act on the hidden state. We will treat the method suite as a record. The type of the object is merely the type of this record. For example, a counter object with an "increment" method and a "read value" method is of type:

type Counter = [inc: Comm, val: Exp Int]

```
COUNTER : Cls Counter
COUNTER = class
             new VAR[Int] x;
             methods
               [inc = (x := x+1), val = x];
             init
                  x := 0
```

Fig. 18. A class of counters.

A *class* describes a particular behavior for objects by giving an implementation, which includes the internal state variables and the definitions of the methods. For example, the class COUNTER of Fig. 18 describes these for counter objects. Having defined the COUNTER class, we can create an instance of this class within a command by writing:

begin new COUNTER c; T end

The counter named c is created at the beginning of the command (by creating its internal state variable and binding c to the method suite) and it is destroyed at the end of the command. We will not consider storable references to objects. So, both the scope and extent of the object c are limited to the command T.

We assume primitive classes

VAR[t]:Cls (Var t)
ARRAY[t]:Int $\rightarrow$ Cls (Var (Array t))

for all data types t. The traditional declaration form var x: t is now equivalent to new VAR[t] x.

Classes, like ADTs, incorporate data abstraction. The difference is that while ADTs export types and expect the client programs to create and manipulate values of such types, classes keep their data representations completely *hidden* from client programs. This is possible because classes work in the context of imperative programming where there is always a hidden mutable state.

For the verification of ADTs in the functional setting, we considered an axiomatic method that relies on axiomatic specifications and an equivalence method that uses simulation relations. The best known method for classes, due to Hoare (23), combines the two techniques, by using axiomatic specifications that incorporate simulation of an abstract representation. These kinds of specifications are often called *model-based specifications.*

To see the issues, let us first consider specifying the behavior of a counter object. If c is an instance of COUNTER then, for all integers k: Int, we have

$\{\text{c.val} = \text{k}\}$ c.inc $\{\text{c.val} = \text{k} + 1\}$

In other words, the effect of c.inc is to change the internal state of the counter in such a way that the value of c.val is incremented. It is possible to specify the behavior of counters directly because the entire state of the object is observable via the val method. However, for more complex data structures, the entire state may not be directly observable. Consider specifying bounded queues with the type shown in Fig. 19. For any integer $\text{n} \geq 1$, QUEUE(n) is a class whose instances denote queues of capacity n. The internal state of the data structure consists of all the elements of the queue, but only the front element is directly observable. We cannot specify the action of, say, the insert operation by its effect on front. The solution then is to consider an abstract

```
type Queue = [init : Comm,
              isempty, isfull : Exp Bool,
              insert : Exp Int → Comm,
              delete : Comm,
              front : Exp Int
             ]
QUEUE: Int → Cls Queue
```

Fig. 19. Type declarations for queue class.

```
(1) {contains(xs) ⇒ q.isempty = (xs = [])}
(2) {contains(xs) ⇒ q.isfull = (length(xs) = n)}
(3) {true} q.init {contains([])}
(4) {not(q.isfull) ∧ contains(xs)} q.insert(x) {contains(xs ++ [x])}
(5) {contains([])} q.delete {contains([])}
(6) {contains(x:xs)} q.delete {contains(xs)}
(7) {contains(x:xs) ⇒ q.front = x}
```

Fig. 20. Axioms for queue class.

representation of queues, say in terms of lists, so that the effect of all the operations can be specified in terms of the abstract representation.

A model-based specification of the queue class is as follows: For all integers $n \geq 1$, and all instances q of QUEUE(n), there exists a parametrized assertion contains: [Int] → assert such that the axioms of Fig. 20 are satisfied. Thus, for every q that is an instance of QUEUE(n), there must be a parametrized assertion contains that relates the state of the queue data structure to a list. The assertion contains(xs) holds in a state if and only if the contents of the queue in that state represents the list of elements xs (with the first element of xs representing the front). For every valid implementation of queues, there must be such a simulation predicate.

The axioms of Fig. 20 are more or less straightforward. Recall that a statement of the form $\{P\}$ means that the assertion P holds in *all* states. So, the first axiom, for instance, says that in any state in which the queue holds the list of elements xs, the boolean expressions q.isempty and xs = [] have the same values. Note that we specify the action of the insert method by its effect on the contains predicate: q.insert(x) changes the state of the queue in such a way that it contains an additional element x at the end.

Consider the queue class shown in Fig. 21 which represents queues by circular arrays. The representation consists of an array of size $n + 1$ (with indices ranging from 0 to n) and two variables f and r to point to the front and rear of the queue, respectively. As a matter of fact, f points not to the front element, but to the position before the front element. The array cell at position f is always unused (called a *dummy cell*).

The methods of a queue object are defined recursively using the operator rec discussed under the heading General Recursion. The recursive definition allows the insert and delete methods to refer to isempty and isfull. To prove that the recursively defined object satisfies its specification, we can use the fixed-point induction principle. However, it is simpler to eliminate the recursion by unfolding the recursive definition once. (This technique works because the recursion used in this definition is benign. It requires only a fixed number of unfoldings.)

To prove that this class meets the specification, we must find a simulation predicate. Let i . . . j denote the sequence of integers i, next(i), next^2(i), . . ., j. Use the notation a[i . . . j] to denote the list of array elements at positions i . . . j. The simulation predicate can then be defined as follows:

```
QUEUE(n) =
 class
  new ARRAY[Int](n+1) a;
  new VAR[Int] f;
  new VAR[Int] r;
  methods
   rec λself.
   [init = (f := 0; r := 0),
    isempty = (f = r),
    isfull = (f = next(r)),
    insert(x) = if not(self.isfull) then
                     begin r := next(r); a[r] := x end
                else skip,
    delete = if not (self.isempty) then
                f := next(f)
             else skip,
    front = a[next(f)]
   ]
    where next(i) = (i+1) mod (n+1);
  init
   begin f := 0; r := 0 end
```

Fig. 21. Queue class using a circular array representation.

contains(xs) $\Longleftrightarrow$
$(f = r \wedge xs = [\,]) \vee$
$(f \neq r \wedge xs = a[\text{next}(f) \ldots r])$

The idea is that the empty queue is represented by the state where f and r are equal (with f pointing to a dummy cell). All other states represent nonempty queues whose elements consist of the elements at positions next(f)..., r. It is now straightforward to verify all the axioms of queues. We show a sample:

- $\{\text{contains}(xs) \to q.\text{isempty} = (xs = [\,])\}$. If contains(xs) is true in a state, then $xs = [\,]$ iff $f = r$, and q.isempty is precisely this condition.
- $\{\text{contains}(xs) \to q.\text{isfull} = (\text{length}(xs) = n)\}$. Suppose contains(xs) is true in a state. If $f = r$ and $xs = [\,]$, then $\text{next}(r) = (r + 1) \bmod (n + 1)$. Since $n \geq 1$, $\text{next}(r) \neq r$. Hence, both isfull and $\text{length}(xs) = n$ are false. If $f \neq r$ and $xs = a[\text{next}(f) \ldots r]$, then lengths(xs) is the same as the number of integers in next(f) ... r. This is equal to n if and only if $\text{next}(r) = f$, which is nothing but the definition of isfull.
- $\{\text{not}(q.\text{isfull}) \wedge \text{contains}(xs)\}$ q.insert(x) $\{\text{contains}(xs \mathbin{++} [x])\}$. We need to show that $\{f \neq \text{next}(r) \wedge \text{contains}(xs)\}$ r := next(r); a[r] := x $\{\text{contains}(xs \mathbin{++} [x])\}$, which amounts to showing that $f \neq \text{next}(r) \wedge$ contains(xs) implies

  $(f = \text{next}(r) \wedge xs = [\,]) \vee$
  $(f \neq \text{next}(r) \wedge xs = a[\text{next}(r) \to x][\text{next}(f) \ldots \text{next}(r)]$

  The first disjunct is impossible. The second follows from the hypothesis.

As in the functional ADTs, the simulation relation for a class is often a function. In that case, we can use an expression instead of a parametrized assertion to model the correspondence with an abstract representation. For example, the following expression for the circular array representation captures the list of queue elements:

```
abs:Exp [Int]
abs = if f = r then [] else a[next(f) . . . r]
```

For other representations, there may also be a representation invariant assertion that specifies which states form valid representations. It is not hard to adapt the axiomatic specification of Fig. 20 to use the invariant and abstraction expression instead.

A good source for the discussion of the abstraction function method is Jones (34). Reynolds (26) and Morgan (36) use the relational method. None of these books deals with objects explicitly. For objects, the articles (39,40) are helpful. They also discuss the issues of subtyping for object-oriented programs.

Conclusion

In this article, we have addressed the subject of programming theory from the viewpoint of ensuring functional correctness of program components. Other major aspects of the subject include *programming language semantics*, which studies general principles of programming language design, the theory of *program specifications*, which studies the specification of large-scale systems, the theory of *concurrency*, which studies techniques for building concurrent and distributed systems, and numerous other theoretical disciplines.

Returning to the issue of functional correctness, we see that there are two major approaches. One is the *semantic* approach, where we use mathematical abstractions to capture the behavior of programs and use them to reason about program behavior. The second is an *axiomatic* or *formal* approach where we use rigorously stated rules to reason about program properties. The two approaches are complementary and the best application of programming theory can benefit from both. The semantic approach better lends itself to intuition and allows one to take large leaps in reasoning. The formal approach generates greater confidence in reasoning, at least if all the steps are carefully followed through. The semantic approach may involve sophisticated mathematical concepts that may be inaccessible without significant effort. On the other hand formal approaches can be applied purely by symbolic manipulations.

The practical application of these theoretical techniques to program development varies widely. In some areas such as protocol design, correctness concerns have a high interest, and systems of small size are even mechanically verified. In some other areas, systems are formally specified using specification languages like Z and VDM. In normal programming, conscientious programmers often document representation invariants for data types so as to aid future modifications. Functional and logic programming languages, whose correctness concerns are simpler than those of imperative languages, have been used for many applications where improved reliability and reduced diff costs have been reported. We anticipate that, in time, theoretical techniques will find wider usage in applications where correctness concerns are critical.

BIBLIOGRAPHY

1. J. C. Mitchell, *Foundations of Programming Languages*, Cambridge, MA: MIT Press, 1997.
2. P. J. Landin, A correspondence between ALGOL 60 and Church's lambda-notation, *Commun. ACM*, **8** (2–3): 89–101, 158–165, 1965.
3. P. Hudak, S. Peyton Jones, P. Wadler (eds.), Report on the programming language Haskell: A non-strict purely functional language (Version 1.2), *SIGPLAN Not.*, **27** (5): Sect. R, 1992.
4. R. M. Burstall, J. Darlington, A transformation system for developing recursive programs, *J. ACM*, **24** (1): 44–67, 1977.
5. R. Bird, P. Wadler, *Introduction to Functional Programming*, London: Prentice-Hall International, 1988.
6. M. C. Henson, *Elements of Functional Languages*, Oxford, UK: Blackwell, 1987.
7. C. A. Gunter, *Semantics of Programming Languages: Structures and Techniques*, Cambridge, MA: MIT Press, 1992.

8. G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*, Cambridge, MA: MIT Press, 1993.
9. R. L. Constable *et al.*, *Implementing Mathematics with the Nuprl Proof Development System*, Englewood Cliffs, NJ: Prentice-Hall, 1986.
10. S. Thompson, *Type Theory and Functional Programming*, Wokingham, England: Addison-Wesley, 1991.
11. R. L. Crole, *Categories for Types*, Cambridge Mathematical Textbooks, Cambridge, UK: Cambridge Univ. Press, 1994.
12. J. Lambek, P. Scott, *Introduction to Higher Order Categorical Logic*, Cambridge, UK: Cambridge Univ. Press, 1986.
13. B. C. Pierce, *Basic Category Theory for Computer Scientists*, Cambridge, MA: MIT Press, 1991.
14. D. S. Scott, A type theoretical alternative to CUCH, ISWIM and OWHY, *Theor. Comput. Sci.*, **121**: 411–440, 1993.
15. S. C. Kleene, *Introduction to Metamathematics*, Amsterdam: North-Holland, 1964.
16. J. Loeckx, K. Sieber, *The Foundations of Program Verification*, 2nd ed., New York: Wiley, 1987.
17. Z. Manna, *Mathematical Theory of Computation*, New York: McGraw-Hill, 1974.
18. J. A. Goguen, Abstract errors for abstract data types, *IFIP Work. Conf. Formal Description Program. Concepts*, 1977.
19. H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification*, Vol. 6, EATCS Monogr. Theor. Comput. Sci., Berlin: Springer-Verlag, 1985.
20. M. Wirsing, Algebraic specification, in J. van Leewen (ed.), *Handbook of Theoretical Computer Science*, Vol. B, Cambridge, MA: MIT Press, 1990, Chap. 13, pp. 675–788.
21. M. Bidoit *et al.*, *Algebraic System Specification and Development: A Survey and Annotated Bibliography*, Vol. 501, Lect. Notes Comput. Sci., Berlin: Springer-Verlag, 1991.
22. R. Milner, An algebraic definition of simulation between programs, *Proc. 2nd Int. Jt. Conf. Artif. Intell.*, London, 1971, pp. 481–489.
23. C. A. R. Hoare, Proof of correctness of data representations, *Acta Inf.*, **1**: 271–281, 1972.
24. J. C. Reynolds, Types, abstraction and parametric polymorphism, in R. E. A. Mason (ed.), *Inf. Processing '83*, Amsterdam: North-Holland, 1983, pp. 513–523.
25. R. Hood, R. Melville, Real-time queue operations in pure LISP, *Inf. Process. Lett.*, **13**: 50–53, 1981.
26. J. C. Reynolds, *The Craft of Programming*, London: Prentice-Hall International, 1981.
27. C. Morgan, T. Vickers (eds.), *On the Refinement Calculus*, Berlin: Springer-Verlag, 1992.
28. J. He, C. A. R. Hoare, J. W. Sanders, Data refinement refined, in B. Robinet and R. Wilhelm (eds.), *ESOP '86, European Symposium on Programming*, Lect. Notes Comput. Sci., Berlin: Springer, 1986, Vol. 213, pp. 187–196.
29. C. A. R. Hoare, J. F. He, J. W. Sanders, Prespecification in data refinement, *Inf. Process. Lett.*, **25** (2): 71–76, 1987.
30. O. Schoett, Behavioral correctness of data representations, *Sci. Comput. Program.*, **14** (1): 43–57, 1990.
31. J. V. Guttag, E. Horowitz, D. R. Musser, Abstract data types and software validation, *Commun. ACM*, **21**: 1048–1063, 1978.
32. C. A. R. Hoare, An axiomatic basis for computer programming, *Commun. ACM*, **12**: 576–583, 1969.
33. D. Gries, *The Science of Programming*, New York: Springer-Verlag, 1981.
34. C. B. Jones, *Systematic Software Development Using VDM*, London: Prentice-Hall International, 1986.
35. E. W. Dijkstra, *A Discipline of Programming*, Englewood Cliffs, NJ: Prentice-Hall, 1976.
36. C. Morgan, *Programming from Specifications*, Hempstead, UK: Prentice-Hall International, 1994, 2nd ed.
37. J. C. Reynolds, The essence of Algol, in J. W. de Bakker and J. C. van Vliet (eds.), *Algorithmic Languages*, Amsterdam: North-Holland, 1981, pp. 345–372 (reprinted as Chapter 3 of Ref. 39.
38. P. W. O'Hearn, R. D. Tennent, *Algol-like Languages.* Boston: Birkhäuser, 1997, 2 vols.
39. P. America, Designing an object-oriented programming language with behavioural subtyping, in J. W. de Bakker, W. P. de Roever, and G. Rozenberg (eds.), *Foundations of Object-Oriented Languages*, Lect. Notes Comput. Sci., Berlin: Springer-Verlag, 1990, Vol. 489, pp. 60–90.
40. B. Liskov, J. M. Wing, A behavioral notion of subtyping, *ACM Trans. Program. Lang. Syst.*, **16** (6): 1811–1841, 1994.

UDAY S. REDDY
University of Illinois at Urbana-Champaign