BUILDING AND EVALUATING ABSTRACT DATA TYPES

A. K. Cline and Elaine Rich

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-83-26 December 1983

Building and Evaluating Abstract Data Types

A. K. Cline
Elaine Rich

The Univeristy of Texas at Austin
Austin, Texas 78712

12 December 1983

Abstract: First a methodology for building an abstract data type is presented and illustrated. This method relies on the abstract modeling technique of specification. The goal of the method is to make the relatively formal use of abstract data types available to programmers as a *practical* tool for program decomposition. The method divides the proof of correctness of an implementation into manageable units. It emphasizes the relationship between the objects of the type (its domain) and its operations. Then several ways of evaluating a type specification are considered. Five *completeness* criteria for type specifications are presented. These criteria provide concrete, syntactic ways of measuring the probable usefulness of a type specification. They guarantee that some necessary relations hold between a type's domain and its operations. For example, the domain should not contain any objects that cannot be created using the operations provided. The domain should contain no distinct objects that cannot be distinguished using the available operations.

CR Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications - Methodologies; D.2.2 [Software Engineering]: Tools and Techniques - Modules and Interfaces; D.2.4 [Software Engineering]: Program Verification - Correctness Proofs; d.3.2 [Programming languages]: Language Constructs - Abstract Data Types;

Table of Contents

1 Introduction

It is by now commonly accepted as wisdom that the only way to build a large, complex program successfully is to split apart the program into a collection of components, each one of which is small enough to be understood and written by a single person in a short period of time. (See, for example, [2].) One way to divide up a large program into smaller pieces is on the basis of the program's control flow. Another way is on the basis of the data objects that the program uses. Abstract data types provide a way of segmenting programs based on this second, data-oriented approach. They provide both a way of segmenting a large program into a set of smaller, more manageable modules, and, simultaneously, a way of defining the necessary interfaces between those modules.

Although a variety of definitions of the the term "data type" have been proposed [6], we will use the following (derived from [11]). An *abstract data type* is a set of values (called the *domain* of the type) and a collection of *operations* that manipulate the elements of the domain. These values and operations are described independently from any commitment as to how they are to be represented inside an actual program, although, of course, a representation must also be provided if the type is actually to be used in real programs.

Figure 1 shows all of the steps required to create a new abstract data type. First the abstract properties of the type must be defined. Then a concrete implementation of the type must be chosen. And finally, a proof that the concrete implementation actually does what the abstract specifications demand must be provided. In the following sections, each of these stages will be discussed in more detail. The process will be illustrated by following one complete type definition through all three stages. The process we describe is similar to that outlined in [15], which in turn is based on the work of [9]. A similar approach was also taken in [4].

2 The Abstract Specifications

The abstract specifications of a data type describe the type without reference to how it will be implemented. The specifications provide a way for a user of a type to reason about the type's behavior in the context of his or her program. Thus they must be formal.

2.1 Stating the Abstract Specifications

There are two ways that such specifications can be provided:

- The first is called the *algebraic* or *axiomatic method* [7, 5, 10]. In this approach, the type is defined from scratch. The domain is specified and a set of axioms that describe the behavior of the operations is also given. This approach has to be used when there is no pre-existing type on which the new type can be based. This approach is very difficult for novices to use correctly. The major difficulty is in producing a set of axioms that is both consistent (i.e. there are no contradictions among the axioms) and complete (i.e. all of the important properties are in fact described in the axioms). It is appropriate to expend the effort required to do this for basic types that will be used often and can then serve as a basis for the definition of new types using the more straightforward abstract modelling approach to be described next. In particular, this approach is often used to define the primitive types that are to be built into a high-level programming language.
- The second approach is called the *abstract modeling method* [15, 12]. In this approach, the domain and the operations are described in terms of the domain and operations of another type that has already been defined. This second type is called the abstract model. The domain specification usually consists of two parts:

Creating a Data Type

1. Provide the abstract specifications
   a. Describe the domain.
   b. Specify the operations.
   c. Show that all elements of the domain can be created given the operations. This is called showing *constructive completeness.*
2. Choose an implementation (concrete representation)
   a. Define the concrete domain.
   b. Define a representation mapping (Repmap) from the concrete domain onto the abstract one.
   c. Provide code for operations.
3. Prove that the concrete implementation meets the abstract specifications
   a. Show that the abstract and concrete domains correspond.
      i. Show that Repmap is a well-defined total function from the concrete domain to the abstract one. In other words, show that all objects in the concrete domain do represent objects in the abstract domain.
      ii. Show that the representation is adequate (i.e. that all elements of the abstract domain have at least one concrete representation).
   b. Show the correctness of each operation.
      i. Show that the operation is correct in the concrete domain.
      ii. Show that the operation correctly implements the abstract specifications.

Figure 1: The Steps in Creating a Data Type

- Source set: a set, often larger than the desired domain, that can easily be described in terms of the type being used as the model.

  Suppose we want to define the type *rational number*. Then the source set is simply the mathematically defined set of rational numbers.

- Invariant: a logical predicate that restricts the actual domain to a subset of the source set. Sometimes no further restriction is necessary. Then we say that the invariant is just T (for true). We will use the notation A(x) to mean that x is an element of the abstract source set and it satisfies the abstract invariant. In other words, it is an element of the abstract domain.

  For the type rational number, the invariant is T. But if we want to define the type *positive rational number*, then we add the invariant

  $x>0$

The operations are defined by stating their pre- and postconditions in terms of operations that have already been defined for the abstract model.

Figure 2 shows a complete set of abstract specifications for the type rational number. These specifications make use of the standard mathematical definitions of operations on rationals.

This abstract modeling technique is a particularly good way to build a collection of useful data types since new types can be defined by building on top of already defined ones. Because of this, and because of the difficulty involved in using the algebraic technique correctly, we will use the abstract modelling approach whenever possible.

The Data Type RATIONAL

```
Domain:
   Source set: the set of all (mathematical) rational numbers
   Invariant: T
Operations:
   function Fraction (x,y: integer): rational;
      pre y ≠ 0
      post RESULT = x/y
   function Equal(x,y: rational): boolean;
      pre T
      post RESULT = (x=y)
   function Ratadd(x,y: rational): rational;
      pre T
      post RESULT = (x+y)
   function Ratsub(x,y: rational): rational;
      pre T
      post RESULT = (x-y)
   function Ratmultiply(x,y: rational): rational;
      pre T
      post RESULT = (x*y)
   function Ratdivide(x,y: rational): rational;
      pre y ≠ 0
      post RESULT = (x/y)
```

Figure 2: Abstract Specifications of the Data Type RATIONAL

Stating the abstract specifications for a new type is a design task. Good specifications are ones that are useful in the context at hand. Because of this, there are very few precise ways to evaluate the merits of a particular set of specifications. But there are a few formal properties that a good set of specifications should possess. These include:

- Constructive completeness - the ability to construct, using the given operations, all of the elements of the domain.
- Testability completeness - the ability to test the preconditions of all of the given operations.
- Discriminatory completeness - the ability to examine elements of the domain and detect differences among them.
- Transformational completeness - the ability to transform each object of the domain into each other.
- Conservation completeness - the ability to perform all the operations required above without side effects on objects of the type.

The first of these is particularly important and will be described here. The other four will be described in Section 5.2.

2.2 Constructive Completeness

The abstract domain is the set of values on which the operations can be performed. In order to use the type, there must be a way for these values to be created. This must be done using the operations that are included as part of the type. For simple types, such as integers, the only necessary operation is usually READ, since that operation can accept as input a character string or other form of every integer that is to be dealt with. But for more complex types, a single operation is usually not sufficient. Instead, a two-stage process is necessary.

1. Some values can be created directly using such operations as READ and CLEAR. These values are called *seeds* and the operations that produce them are called *seed generators*.

2. The others must be built by successive applications of other operations applied initially to one of the seeds. The operations used for this are called *inductive generators*.

For example, to create all of the elements of the domain of the type stack of integers, it is sufficient to have access to a function CLEAR (which creates one seed, the empty stack) and a function PUSH (which takes any existing element and creates a new one). If PUSH were omitted, then the type would not be *constructively complete*, since not all elements of the domain could, in fact, exist. Notice that constructive completeness is independent of implementation. Any implementation of CLEAR and PUSH that meets the specifications will suffice to guarantee that all objects of the domain can be created.

For the type rational number, only the first step is necessary to show constructive completeness. All rationals can be created directly using the Fraction function.

In some other treatments of this subject (e.g. [15]) there is a distinguished initialization operation that produces a distinguished initial value. We have substituted the set of seed generators for this because there is no logical reason why there should be a single initial value. For structured types, such as stacks and sets, such a value is meaningful. But particularly for unstructured types, such as integers, there is no such value. The key part of an initialization operation, namely that initializing operations must create objects in the domain of the the type without depending on being provided any such objects as input is retained here and will reappear in Section 4 in which we discuss the proofs that all operations are correctly implemented.

Constructive completeness is important for three reasons:

- It provides a way of measuring the abstract specifications to see how useful the type they define is likely to be.

- It forces a definition of the abstract domain that does not include superfluous objects that can never in fact exist but which may limit the choices of implementation if they must be accounted for.

- It reduces to trivial the proof that a concrete representation is adequate. This will be discussed more later.

If a type cannot be shown to be constructively complete, its specifications can be modified in one of two ways:

1. Restrict the domain to only those elements that can be constructed.

2. Provide additional operations so that all elements of the domain can be constructed.

The choice between these two must depend ultimately on what the type is to be used for.

It is interesting to note that almost all types either possess a single seed (e.g. set, stack) or all elements

of the type are seeds (e.g. integers). One class of types that constitute exceptions to this rule are those defined by recurrence relations of order greater than one. For example, to define Fibonacci numbers, two seeds are necessary.

3 The Concrete Implementation

In order for an abstract type to be useful, it must be implemented using some combination of primitive machine structures and other types and operations that have already been defined. Often, there is more than one reasonable implementation. It may be necessary to describe several and analyze their performance. The choice among them may then be obvious, or it may depend on how the type is going to be used.

The first thing that must be done in defining a concrete implementation of a data type is to define the concrete domain. This is usually done in two stages, just as it is at the abstract level:

- Define the source set. This is done with a type declaration in the programming language that is being used. This declaration may of course build on other, already defined types.

  A source set definition for the type rational number is

  ```
  type rational = record numerator: integer,
                         denominator: integer   end;
  ```

- Provide an invariant that restricts the actual domain to only those values that satisfy particular constraints, which may be imposed by space restrictions or by the way that some of the operations will be performed. For example, it might be desirable to keep a vector of values sorted so that binary search can be used to locate desired objects. Just as at the abstract level, the invariant may be simply T. We will use the notation C(y) to mean that y is an element of the concrete source set and it satisfies the concrete invariant. In other words, it is an element of the concrete domain.

  For this concrete representation of rational numbers, the simplest invariant we could use is

  $y.\text{denominator} \neq 0$

  We might, however, choose a more restrictive invariant. For example, we might require

  $y.\text{denominator} \neq 0 \wedge$
  (y.numerator and y.denominator have no common factors)

  This restricted invariant might be used in order to simplify the procedure that checks for equality.

Notice that the domain is described in two parts. For an object to be in the domain, it must both belong to the source set and satisfy the invariant. This is a slightly different formulation than has been used before (e.g. in [15]), in which the domain is defined as the source set and the invariant is treated separately. This separate treatment arises somewhat naturally from the fact that the syntax of the language in which the implementation is written typically guarantees that objects belong to the desired source sets, while a semantic verification method must be used to guarantee that an invariant is satisfied. We will make use of this syntactic guarantee in our correctness proofs in Section 4. But it is important to note here that the domain specification does consist of these two parts and only objects that satisfy both need be considered.

The next thing that must be done in defining a concrete implementation is to define each of the required operations in code. This is done in the usual manner.

The third thing that must be done to define a concrete representation is to specify a mapping from objects in the concrete domain to objects in the abstract domain. This mapping function will be called the *representation mapping* or Repmap, and will be denoted as R(y). Figure 3 illustrates the relationship between the abstract domain, the concrete domain, and Repmap.

For rational numbers, we have

```
R(y) = y.numerator/y.denominator
```

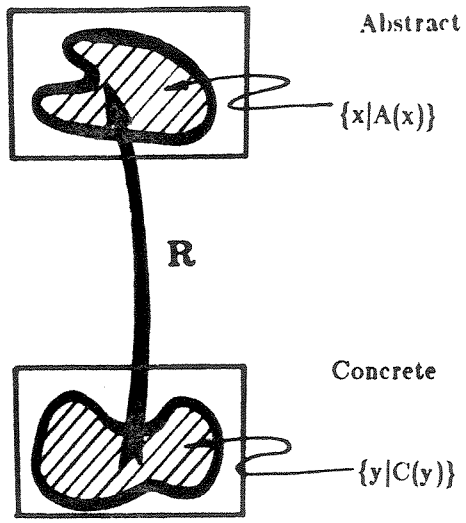

Figure 3: Repmap Maps Concrete Elements into Abstract Ones

Throughout the rest of this paper, we will use the variables x and u to represent objects in the abstract domain and y and z to represent corresponding objects in the concrete domain. In particular, assume that R(y) yields x and R(z) yields u.

Repmap must have two properties:

- It must be a well-defined total function. In other words, every object in the concrete domain must map to an object in the abstract domain. If this were not the case, then the result of a concrete operation might make no sense at the abstract level. This is illustrated in Figure 4. One of the roles of the concrete invariant is to restrict the concrete domain enough that this property is satisfied.
- It must be onto. In other words, every object in the abstract domain must have at least one concrete object that maps to it. If this were not the case, then some abstract objects would have no representation. In this situation, the representation is said not to be adequate. Figure 5 illustrates this situation.

Repmap need not, however, be one-to-one. Sometimes one abstract object will have many concrete representations. Figure 6 illustrates this.

For example, if we use the weaker invariant discussed above, then (1,2), (2,4), and (7,14) are all representations of the same rational number, 1/2. If we want to force Repmap to be one-to-one, we can do so by using the stronger invariant (i.e. require that every ratio be expressed in lowest terms). For the rest of this paper, though, we will use the weaker invariant that does not include this restriction.

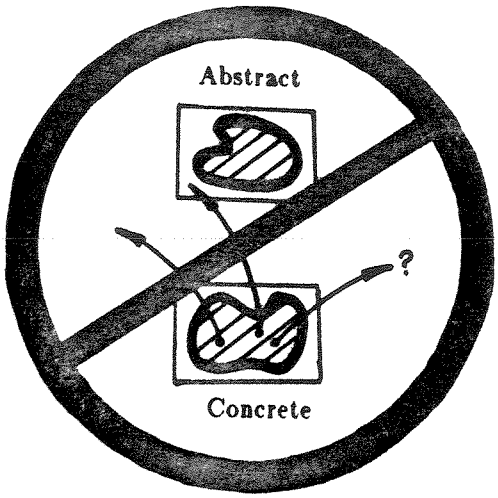

Figure 4: Repmap Must be a Total Function

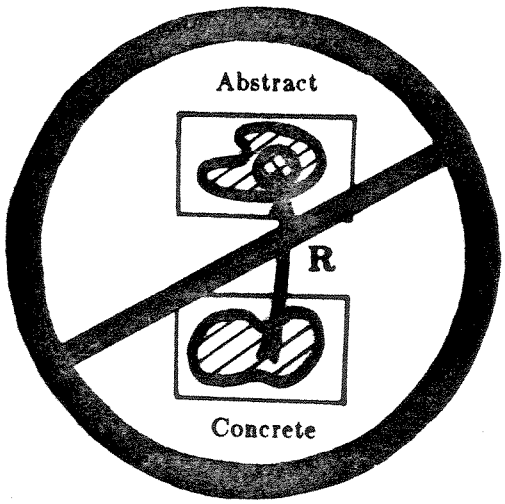

Figure 5: The Representation Must be Adequate

4 Prove that the Concrete Implementation Corresponds to the Abstract Specifications

An abstract data type must actually behave as its abstract specifications stipulate. But, in fact, its behavior is determined not by its abstract specifications but by its concrete implementation. Thus it is important to be able to prove that the concrete implementation corresponds to the abstract specifications. By this we mean that all of the properties described at the abstract level are guaranteed by what happens at the concrete level. The process of constructing this proof consists of two parts:

1. Show that the abstract and the concrete *domains* correspond to each other.

2. Show that the abstract and concrete *operations* correspond to each other.

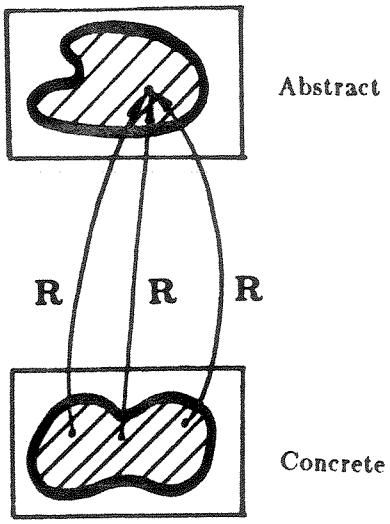

Figure 6: Repmap Need Not Be One to One

4.1 Matching the Domains

Showing that the abstract and the concrete domains match is a two-step process. The first is to show that Repmap is well-defined. The reason that this is important was described above, and need not be repeated here.

> In the example, notice that Repmap is defined exactly so long as y.denominator is not 0. The concrete invariant guarantees that. In fact, that is why y.denominator $\neq 0$ is the weakest concrete invariant we could use.

The second step of the proof process is to show that the representation is adequate. In other words, we must show that, for every object in the abstract domain, there exists at least one corresponding (as defined by Repmap) object in the concrete domain. Constructive completeness guarantees that, at the abstract level, every object can be constructed using the operations that are provided. Thus, if those operations are implemented correctly, the corresponding objects will be created properly at the concrete level unless restrictions at the concrete level prevent the operations from being applied. These restrictions could come either from:

- A weak concrete domain that prevents some seeds from being constructed. This usually happens if the concrete invariant is too strong (restrictive). But if this occurs, it will not be possible to prove the correctness of the seed generators (such as CLEAR or READ) since these operations must produce objects in the concrete domain.
- A strong concrete precondition on required inductive generators such as PUSH. But if this is the case, we will not be able to prove that the concrete PUSH correctly implements the corresponding abstract operation. This will be explained below.

Thus it is not in fact necessary to show specifically that a concrete implementation of an abstract type is adequate. Abstract completeness and the correctness of all seed generators and inductive generators guarantees adequacy. Figure 7 shows the logical relationships between constructive completeness, concrete constructive completeness (parallel to constructive completeness but at the concrete level), adequacy, and the correctness of the generating operations (both seed-producing and inductive). The arrows indicate implication. Joined arrows indicate logical $\wedge$.

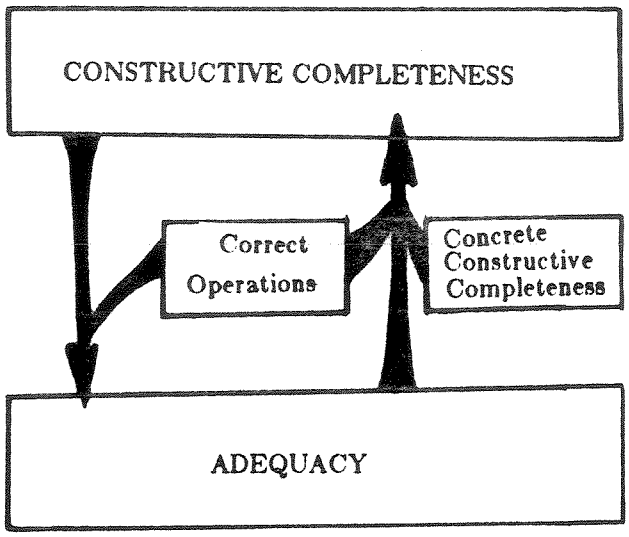

Figure 7: The Logical Relationship between Completeness and Adequacy

4.2 Matching the Operations

After showing that the abstract and the concrete domains match, it is necessary to show that each abstract operation is correctly implemented at the concrete level. Each abstract operation OP_A will actually be simulated using a concrete operation OP_C.

Consider the abstract operation of division of two rationals. Given two rational numbers x and u, it should produce the rational number x/u. The code to simulate that at the concrete level is the following:

```
function ratdivide(y,z:rational):rational;
     begin
        ratdivide.numerator := y.numerator * z.denominator;
        ratdivide.denominator := z.numerator * y.denominator
     end;
```

To simplify the proof procedure, we will attach to each concrete operation a concrete precondition and a concrete postcondition (as is done in [15].) Doing this makes it possible to prove the correctness of the code by referring solely to objects in the concrete domain. Then, in separate steps, we prove that the concrete and abstract preconditions "match" as do the concrete and abstract postconditions. We may choose any concrete pre- and postconditions that make it possible both to show concrete correctness and to show the desired match with the abstract domain. If the concrete operations are correct, then there will exist at least one (but possibly more than one) pre- and postcondition pair that makes this possible.

Figure 8 illustrates the relationship between the abstract operations, the concrete ones, and Repmap. To simplify this picture as well as later ones, we will show operations that map one object of a given type into another object of the same type. Of course, this is not the only kind of operation that is allowed. If the result type is different from the argument type, then the R's in the left side of the figure will refer to a different representation mapping than those on the right and the domains (both abstract an concrete) on the left will be different from those on the right. If more than one argument is allowed, or if more than one value is computed, then the domains must in fact represent cross-products of domains of defined types.

For example, at the abstract level, ratdivide maps (1/2, 3/2) into 1/3. At the concrete level, ratdivide must map any pair of records where the first represents 1/2 and the second represents 3/2 into a record that represents 1/3.

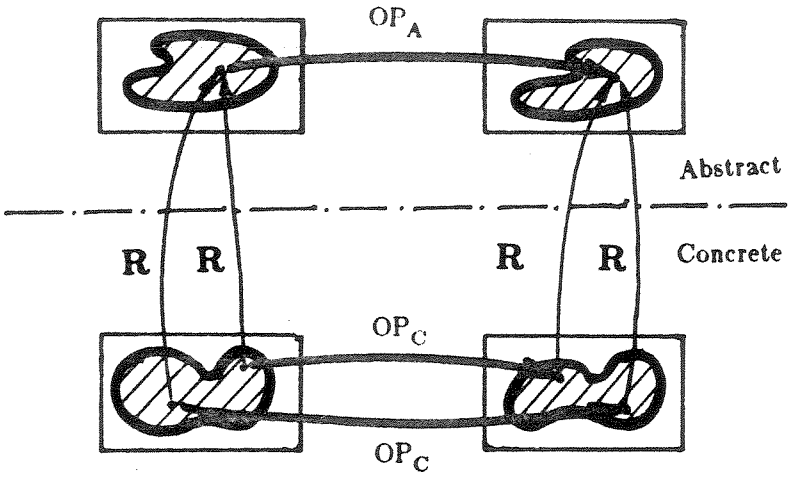

Figure 8: Simulating Abstract Operations with Concrete Ones

4.2.1 Step 1: Mapping Preconditions

The set of objects upon which an operation may be performed contains exactly those domain elements that satisfy the operation's preconditions. We shall denote the precondition of the abstract operation as AbstPre and the precondition of the concrete operation as ConcPre. Figure 9 shows a graphic representation of the role of preconditions.

The precondition for the abstract operation of dividing two rational numbers, x by u, is that u not equal 0. The precondition for the corresponding concrete operation is that z.numerator not equal 0.

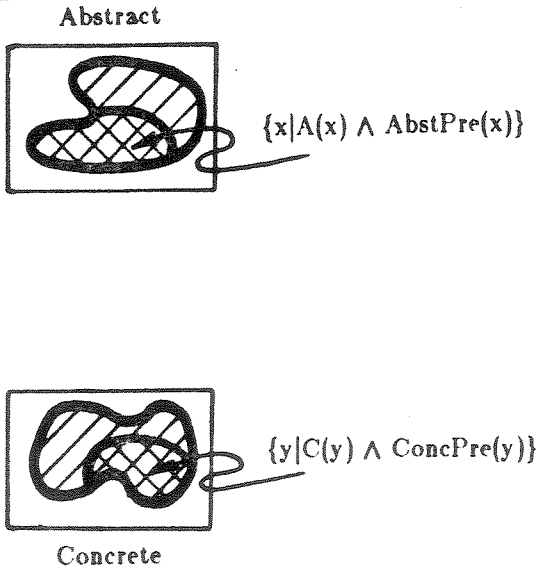

Figure 9: Preconditions Define the Domain of an Operation

Since the abstract operation will be simulated by the application of the concrete operation to concrete representatives of the abstract elements, we must guarantee that the concrete preconditions are sufficiently weak that they will be satisfied by *any* representative of any element of the abstract domain satisfying the abstract preconditions. Thus, what we must have is:

$$\forall x\ A(x) \wedge AbstPre(x) \Rightarrow (\exists y\ C(y) \wedge x{=}R(y) \wedge ConcPre(y)) \wedge (\forall y\ C(y) \wedge x{=}R(y) \Rightarrow ConcPre(y))$$

This says that for every abstract object satisfying the abstract precondition:

- There exists at least one corresponding concrete object satisfying the concrete precondition, and
- All corresponding concrete objects (since there may be more than the one that is required) satisfy the concrete precondition.

However, since we have proved that R is a total function, we know that

$$C(y) \Rightarrow A(R(y))$$

Using this and the fact that the representation is adequate, it is possible to rewrite the previous formula more concisely as

$$\forall y\ C(y) \wedge AbstPre(R(y)) \Rightarrow ConcPre(y)$$

In other words, all concrete objects that represent abstract objects satisfying the abstract precondition must satisfy the concrete precondition. In terms of sets, we have

$$\{y | C(y) \wedge AbstPre(R(y))\} \subseteq \{y | ConcPre(y)\}$$

But it is not necessary to guarantee explicitly that y is an element of the concrete source set, since this is guaranteed already by the syntactic typing mechanism of the language in which the declaration of the source set occurs. All that must be dealt with explictly is the concrete invariant. Thus what we will actually prove is

$$\forall y\ ConcInv(y) \wedge AbstPre(R(y)) \Rightarrow ConcPre(y)$$

We shall call proving this the first step in verifying an operation.

To perform this step for our example is easy. We must show that

$$\forall y\ \forall z\ (y.denominator \neq 0 \wedge z.denominator \neq 0 \wedge R(z) \neq 0) \Rightarrow z.numerator \neq 0$$

This is obvious, since if z.numerator were 0 then z would represent the abstract object 0.

Notice that we must have *every* representative of an abstract object that satisfies the abstract precondition satisfy the concrete precondition. This is important since we cannot guarantee that, at some moment when the operation is about to be performed, the current representative of the abstract element satisfies the concrete preconditions simply because some representative does. Figure 10 shows an example of a situation that must not occur.

In our example, notice that if (for some reason) the concrete operation had had as its precondition

$y.numerator \neq 0$

we would not have been able to prove this step. In such a case, the concrete operation would not be sufficiently general to simulate the abstract operation for all legal data. We would not be able to divide zero by anything. Or if the concrete precondition had been

$z.numerator \neq 0 \wedge z.denominator \neq 1$

then every acceptable x and u would have *some* acceptable representatives. However, this is not sufficient since we cannot guarantee that the current representative will always be an acceptable one. For example, if 3 happens to be represented as 3/1, it could not be used, but if it is represented as 6/2 it could.

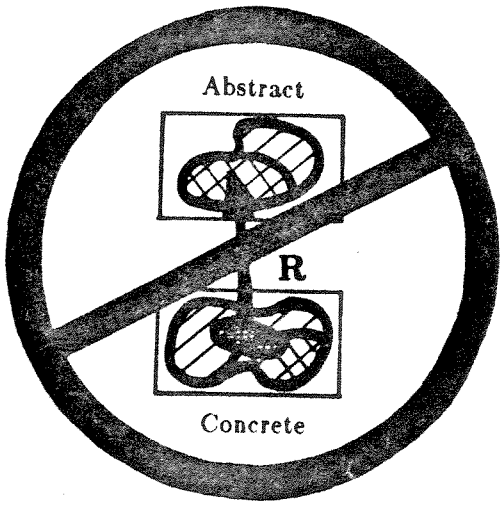

Figure 10: All Representatives Must Satisfy ConcPre

4.2.2 Step 2: Correctness at the Concrete Level

Next we must prove that the concrete operation is correct simply at the concrete level. To do this, we must consider not only the production of the desired postconditions but also the preservation of the concrete invariant. Without this, the operation might produce concrete objects that are not representatives of any abstract objects. Figure 11 illustrates this problem. Of course, since we only perform the operation on elements satisfying the concrete invariant, we are allowed to assume this in our correctness proof. The one exception to this is that operations that are used to initialize objects cannot assume that the concrete invariant holds when they begin. More precisely, the seed generators that are used in the first step of the proof of constructive completeness may not assume the concrete invariant since these are the operations that directly create new values of the type. For noninitializing operations, though, what we must prove is that

$$\{ConcPre(y) \wedge C(y)\}$$

$$OP_C$$

$$\{ConcPost(y) \wedge C(y)\}$$

ConcPost denotes the concrete postcondition. We shall call proving this step two of the verification

process. Notice that we are exploiting the proof rule of *generator induction* [14] to prove that the concrete invariant holds for all objects that might be treated as elements of the concrete domain.

The concrete postcondition for our example function is

```
(RESULT.numerator=y.numerator*z.denominator ∧
 RESULT.denominator= y.denominator*z.numerator)
```

Thus the second step of the verification process requires that we prove

```
{z.numerator≠0 ∧ y.denominator≠0 ∧ z.denominator≠0}

     ratdivide.numerator:=y.numerator*z.denominator;
     ratdivide.denominator:=z.numerator*y.denominator

{RESULT.numerator=y.numerator*z.denominator ∧
 RESULT.denominator= y.denominator*z.numerator ∧
 RESULT.denominator ≠ 0}
```

The proof of this is easy to do using weakest preconditions [3].

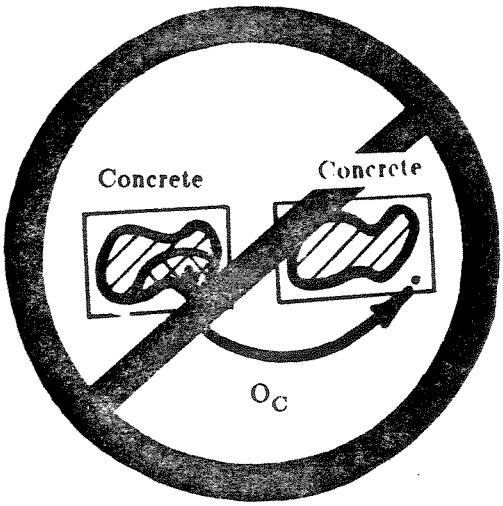

Figure 11: The Necessity of Maintaining the Invariant

4.2.3 Step 3: Mapping the Postconditions

After these first two verification steps, we know that the operation takes all concrete representatives of all the appropriate abstract objects and produces objects satisfying the concrete postcondition. We do not yet know, however, if these objects are the representatives of the desired abstract objects, namely those satisfying the abstract postcondition. Thus the third and last step of verifying an operation is to show

$$\forall y\ C(y) \wedge \text{ConcPost}(y) \Rightarrow \text{AbstPost}(R(y))$$

AbstPost denotes the abstract postcondition. Notice the direction of the implication. We insist that the concrete postcondition be strong enough to guarantee the abstract postcondition. It is allowed for it be be far stronger but it is not allowed for it to be any weaker. This contrasts with [1], in which the two postconditions must be semantically identical. In terms of sets, we have

$$\{R(y) \mid C(y) \wedge \text{ConcPost}(y)\} \subseteq \{x \mid \text{AbstPost}(x)\}$$

Again we remove the explicit statement that y must be an element of the concrete source set since that is guaranteed syntactically. Thus what we must actually prove is

$\forall y$ ConcInv(y) $\wedge$ ConcPost(y) $\Rightarrow$ AbstPost(R(y))

The abstract postcondition of divide is, of course,

RESULT = x/u

Thus the third step of the verification requires that

$\forall y \; \forall z$ (RESULT.denominator $\neq 0 \wedge$
RESULT.numerator=y.numerator*z.denominator $\wedge$
RESULT.denominator=z.numerator*y.denominator)
$\Rightarrow$ R(RESULT) = R(y)/R(z)

This is obvious from the definition of Repmap.

4.2.4 Summary

These three steps are illustrated in Figure 12. The arrows in the figure should be read as implication symbols. Our real goal is to show that if OP_A is applied and the conditions of Boxes A_1 and A_2 hold, then the conditions of Box D can be guaranteed to hold. But we cannot do that directly, since only concrete operations can actually be executed. Instead, we prove that the conditions in Boxes A_1, A_2, and B_2 imply those in Box B_1 (Step 1). Then we show that the conditions in Boxes B_1 and B_2 imply those in Box C given the operation OP_C (Step 2). And finally, we prove that the conditions of Box C imply those of Box D (Step 3). We also appeal to our demonstration that the concrete representation is adequate to guarantee that every abstract object that satisfies the conditions of Boxes A_1 and A_2 has a concrete representative that satisfies the conditions of Box B_2.

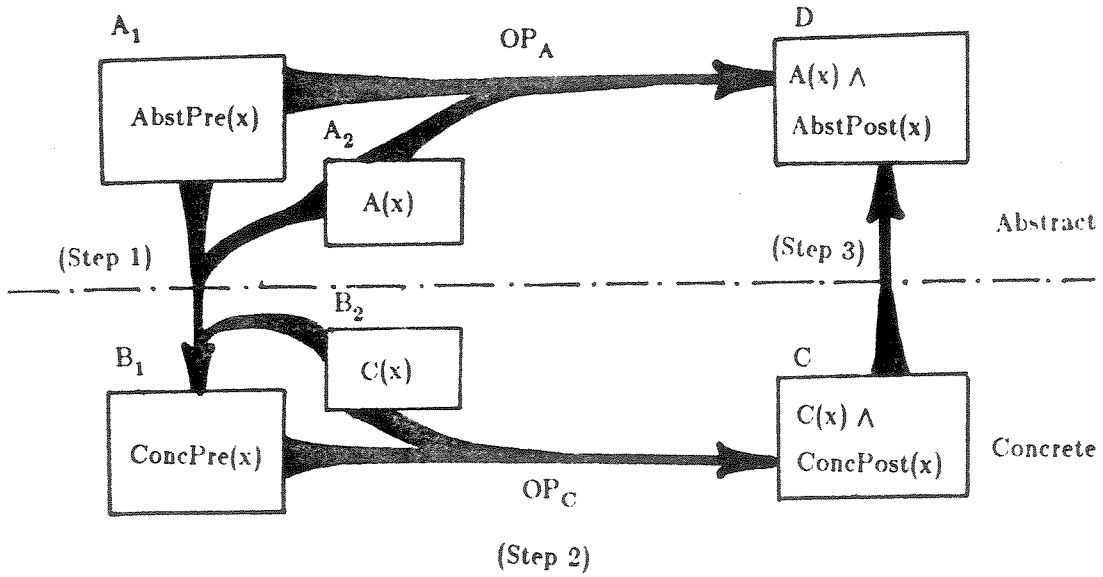

Figure 12: The Three Steps for Verifying an Operation

So far, we have described the process of proving the correctness of an operation as a process of finding some concrete pre- and postcondition that make all three steps provable. Steps 2 and 3 as they have so far been presented do, however, represent stronger implications than are really required and so the choice of concrete pre- and postconditions has been more restricted than necessary (with the benefit, though, that the implications are easier to work with). It is worth pointing out, though, the more restrictive forms of the implications:

Step 2:
$\forall y\ C(y) \wedge AbstPre(R(y))\ \{OP_c\}\ ConcPost(y) \wedge C(y)$

Step 3:[1]
$\forall y'\ ConcInv(y) \wedge AbstPre(R(y')) \wedge ConcPost(y) \Rightarrow AbstPost(R(y))$

To see what is happening here, consider the following example.

Suppose Ratadd had the following abstract specifications:

```
function Ratadd(x,y: rational): rational;
   pre x>0 ∧ y>0
   post RESULT = (x+y)
```

Further suppose that this operation were implemented by summing the absolute values of the inputs and that the concrete pre- and postconditions were

```
pre T
post RESULT = |x| + |y|
```

Assume that Step 2 has been done. (Step 1 is trivial, since anything implies T.) Step 3 fails, however, using the old rule. This postcondition is too weak to guarantee the abstract postcondition for arbitrary rational numbers x and y. But for any x and y that satisfy the abstract precondition, this concrete postcondition does guarantee the abstract one, and so the proof can be completed using the modified rule for Step 3.

Alternatively, the concrete pre- and post conditions can be changed to

```
pre x>0 ∧ y>0
post RESULT = (x+y)
```

Step 1 can still be done. So can Step 2 if it could be done before, since the new postcondition is equivalent to the old one whenever the precondition is satisfied. And now Step 3 can be done using the old rule.

5 Evaluating an Abstract Specification of a Data Type

Up to this point, we have described a way in which a set of abstract specifications for a data type may be provided, a way in which a concrete implementation may be specified, and a way of demonstrating that the two correspond. But we have said very little about how abstract specifications or concrete implementations can be evaluated to determine if one is, in some sense, better than another. For concrete implementations, there exists a substantial body of literature. For example, one important property of a concrete implementation is efficiency, which can be evaluated using standard techniques for program analysis. (See, for example, [13].) For abstract specifications, on the other hand, the situation is much less clear-cut. In this section we will describe some criteria, however, by which an evaluation can be made.

5.1 Correctness

Recall that in evaluating concrete implementations, the one property that we absolutely insist upon is correctness. This suggests that we should perhaps insist on the same property for an abstract specification. The question then arises of what it would mean for an abstract specification to be correct. Two things are possible:

---

[1] In this and later examples, prime notation will be used to refer to the initial value of a parameter when it is necessary to distinguish that from the output value.

- The specification is internally consistent.
- The specification corresponds to some other model of the type.

It is, of course, imperative that the abstract specification be consistent. When the algebraic approach to specification is used, statments are made about the relationships among the operations and it is these statments that must be consistent. Showing that they are is often difficult. But with the abstract modeling approach, the only statements that are made involve the relationships between the pre- and postconditions of individual operations. Thus to show the consistency of a specification, it suffices to show the consistency of each operation individually. The specification of a particular operation OP is inconsistent if and only if

$$\exists x \neg(\text{AbstPre}(x) \Rightarrow \text{AbstPost}(\text{OP}(x)))$$

The simplest example of an inconsistent specification is the following:

```
pre T
post F
```

A less blatent example is:

```
pre x<0
post x=x' ∧ x>0
```

But if a specification is inconsistent in this sense, there can exist no corresponding implementation. Thus it will not be possible to show that any implementation is correct, since such a correctness proof would require showing that for all objects satisfying the abstract precondition the abstract postcondition is guaranteed by the concrete operation. Thus the verification method described in Section 4 has the side effect of proving that the abstract specification it is given is consistent.

The second way in which the correctness of a specification can be defined is with respect to some other, previously defined model [5]. This is particularly useful for types, such as integers, for which such a model already exists. But many of the types that programs need do not already possess another formal model. For some, though, there may exist statements that can be made about desired behavior and it is useful to prove that the type, as specified, corresponds to those statements. For example, for a stack, PUSH and POP should be inverse operations. This can be guaranteed by showing that

```
{T}

PUSH(s,x);
POP(s);

{s=s'}
```

This can easily be done given, for example, the stack specification shown in Figure 13.

Consistency correctness is important, but it is weak. There exist many specifications that are consistent but useless. The effectiveness of showing a correspondence with another model or with a set of independent assertions is limited by the availability of such a model or set of assertions. In the absence of any stronger notion of correctness, some other notion of usability is required. In the next section, such a notion will be developed.

5.2 Completeness

In this section, we try to describe properties that measure the usefulness of an abstract specification of a data type. All of these properties can be described as *completeness* properties since they evaluate the scope of various aspects of the specification.

There are two levels at which one can talk about the completeness of an abstract data type sepcification:

- The completeness of the set of objects and operations provided in the specification.
- The logical completeness of the *description* of the behavior of those objects and operations.

Other work in this area [8] has focused on the second of these, which is a well-defined problem when the algebraic technique is used to define the semantics of a type. When the abstract modeling technique is used, however, we normally assume that all properties of interest are stated in the postconditions of the operations and that anything left unspecified is not of concern.

In this discussion, we intend to focus instead on the first type of completeness in which we evaluate not the description of the type but rather the type itself. In other words, we may ask whether enough of the right(i.e. useful) operations have been provided rather than how detailed the descriptions of the operations are.

In Section 2, a set of completeness properties that may hold for an abstract specification of a data type was outlined. These included:

- Constructive completeness
- Testability completeness
- Discriminatory completeness
- Transformational completeness
- Conservation completeness

Constructive completeness has already been defined since it interacts with the proof that a particular concrete representation is adequate with respect to a particular abstract specification. We will soon elaborate on it and describe the other four.

But before we do that, we should ask, "In what specific ways should completeness properties be able to measure the usefulness of a data type?" The answer is the following:

- The definition of an abstract data type should not place unnecessary constraints on the concrete implementation of the type since doing so could preclude the use of a highly efficient technique.
- The definition of an abstract data type should provide a large enough collection of operations that it will not be necessary for a programmer to develop a new type that is a close cousin of the first one but that allows more flexibility in manipulating objects of the type. The existence of families of separate types that arise for this reason obscures much of the clarity that can be gained by using types as an abstraction mechanism.

5.2.1 Constructive Completeness, Again

Constructive completeness has already been defined informally. It can be defined formally as follows:[2]

[2]We use functional notation here for convenience. If operations with side effects are involved, then this definition must be rewritten to use statement sequencing rather than function composition to define a series of computations.

$\forall x\ A(x) \Rightarrow \exists p_1 \ldots p_n \mid p_n[\ldots p_1] = x$
where p_1 is either a constant function or a function with arguments
of a type other than the one being defined.

Constructive completeness is a property of the relationship between the domain of a type and the operations that the type provides. Recall from Section 2.2 that if a type could not be shown to be constructively complete, then either the domain or the operations could be changed to make it so. But can both of these always be done? Specifically, are there some domains that are inherently constructively incomplete, independent of the set of operations provided? Clearly if we restrict our attention to operations that are computable (which we must do eventually if we are to provide a concrete implementation) then the answer is yes, since there exist sets that are not recursively enumerable. An example of such a set is the real numbers. The only way to define a constructively complete type corresponding to such a set is to constrain the domain (such as with a precision restriction).

5.2.2 Testability Completeness

If the specifications for abstract operations contain preconditions that must be checked by callers of those operations, then functions must be provided to enable that checking to be done. A set of abstract specifications in which all preconditions can be checked, either by functions provided within the specifications themselves or in the specification of other data types that will be available whenever the new type is used, is said to possess *testability completeness*. Without this property, some specified operations (namely those whose preconditions cannot be checked) cannot be called and so are useless. Yet they must be provided in any correct implementation.

> In the specifications for rational numbers, only two operations have preconditions that are not simply T. Fraction requires that its second integer argument not be zero. This can be checked using the equality predicate defined on the integers. Ratdivide requires that its second rational argument not be zero. This can be tested with the function Equal that is given in these specifications for rationals.

Testability completeness often requires the use of operations defined on objects of types other than the one being defined. For example, it is possible to test the precondition

SIZE(X) < MAXSIZE

using the following operations:

- SIZE, which is defined on objects of the type being defined and which returns an integer.
- MAXSIZE, which is an integer constant that must be defined as part of the type being specified.
- <, which is defined on the integers and which returns a boolean value.

Thus the result of a check for testability completeness is not just a boolean value but rather is a list of required functions outside of the current definition. The type being defined will then possess testability completeness in any definition environment in which the required auxiliary functions are defined.

Of course, another way to test this same precondition is with a single function defined as follows:

```
function preok?(x: t): boolean;
   post RESULT = (size(x) < maxsize)
```

Although testing preconditions in specialized functions such as this satisfies testability completeness, it is generally a bad idea since it fails to make available to users of the type component operations that may be independently useful. This is important because preconditions often arise naturally from the logical

structure of the objects being manipulated. So the conditions they mention are often important properties of those objects. An extreme example of this is the following excerpt from a bad specification for the type SET:

```
procedure INSERT(var s: set; x: integer);
   pre cardinality(s ∪ {x}) ≤ max
   post s = s' ∪ {x}
function OKTOINSERT(s: set; x: integer): boolean;
   post RESULT = (cardinality(s ∪ {x}) ≤ max)
```

A better way to test the precondition for INSERT is to provide the operations SIZE and MEMBER?.[3]

It is interesting to note that testability completeness is the only one of the completeness properties we discuss that is not monotonic in the number of operations provided. A type that possesses testability completeness may lose it if a new operation with an untestable precondition is added.

5.2.3 Discriminatory Completeness

Figure 13 shows an abstract specification for the type STACKOFINT.[4] This definition corresponds to the standard idea of how a stack should behave. Suppose now that we remove the operation TOP, leaving everything else the same. Intuitively, this new type is much less useful than the old one. Why is this? The answer can be seen by looking at the two stacks shown in Figure 14. Both can be created using the operations provided. But, using those operations, it is not possible to detect any difference between the two.

The Data Type Stack of Integers

```
Domain:
   Source set: the set of sequences of integers
   Invariant: T
Operations:
   procedure CLEAR(var s:stack);
      post s=<>
   procedure PUSH(var s:stack; x:integer);
      post s=<x> ~ s'
   procedure POP (var s:stack);
      pre s ≠ <>
      post s = trailer(s')
   function TOP(s:stack):integer;
      pre s ≠ <>
      post RESULT = first(s)
   function EMPTY?(s:stack):boolean;
      post RESULT = (s=<>)
```

Figure 13: Abstract Specifications of the Data Type STACKOFINT

Since no operations can detect the difference between the two stacks, we say that the type does not possess *discriminatory completeness*, which can be defined formally as follows:[5]

$$\forall x,y\ A(x) \wedge A(y) \wedge x \neq y \Rightarrow \exists p_1,\ldots,p_n \mid p_n \text{ is a boolean function}$$
$$\wedge\ p_n([p_{n-1}(\cdots p_1(x))],[p_{n-1}(\cdots p_1(y))])$$
$$\wedge\ \neg\ p_n([p_{n-1}(\cdots p_1(x))],[p_{n-1}(\cdots p_1(x))])$$

---

[3] where SIZE(s) returns the cardinality of s and MEMBER?(s,x) returns true iff x is an element of s.

[4] This specification is modeled on the type, sequence, defined in, for example, [16]. The symbol ~ stands for concatenation.

[5] We again use functional notation here for convenience.

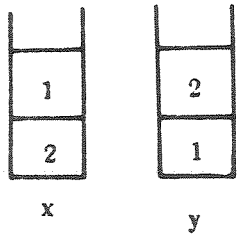

Figure 14: Two Stacks

In other words, for any two distinct objects in the domain, there exists a sequence of operators that detects that the objects are different. P_n is usually the function $\neq$. Recall that in showing testability completeness it may be necessary to use operations on types other than the one being defined. Thus the form of the completeness statement was that a type is complete in any environment containing the required list of auxiliary functions. The same situation occurs here as well.

To see how this requirement works, we will consider an example. Return, for the moment, to the complete definition for STACKOFINT given in Figure 13. Then for the two stacks shown in Figure 14 we get

$$\neq(\mathrm{TOP}(x),\mathrm{TOP}(y))$$
$$\wedge$$
$$=(\mathrm{TOP}(x),\mathrm{TOP}(x))$$

In this example, we have used two operations, TOP (defined as part of the type STACKOFINT) and $\neq$ (defined on the integers). In fact, it is easy to show that any two different stacks can be seen to be different using the operations POP, TOP, and EMPTY? defined on STACKOFINT, $\neq$ defined on the integers, and $\neq$ defined on the type boolean. If two stacks have different lengths, then a series of POPs followed by EMPTY? followed by boolean $\neq$ will detect the difference. If two stacks have the same length but at least one different element, a series of POPs followed by a TOP and an integer $\neq$ will detect the difference.

Notice, by the way, that we require only that there exists a sequence of operations that will detect the difference, not that there exist a program that uses only the given operations and that finds the difference. Such a program would, of course, if it existed, implement the $\neq$ function for STACKOFINT.

Suppose now that we again remove the function TOP from the specification. Now the type does not possess discriminatory completeness. Two stacks of different lengths can be seen to be different but not two stacks of the same length but containing different values.

If a type cannot be shown to possess discriminatory completeness, its specifications can be modified in one of two ways:

1. Restrict the domain to only those elements that are unique with respect to the accessing operations that are provided.

2. Provide additional operations so that all elements of the domain behave distinctly.

Notice that these alternatives parallel those described in Section 2.2 for producing constructive completeness.

Consider again the example of STACKOFINT minus TOP. Either stack behavior is desired, in which case action 2 should be taken and TOP should be added, or what is really desired is a counter, in which

case the domain should be redefined as described in action 1. The new domain will contain one element for each equivalence class of elements of the old domain under the equivalence relation "cannot be told apart from". In this case, the new domain wll be the non-zero integers, each of which corresponds to the length of the stacks in one of the equivalence classes.

5.2.4 Transformational Completeness

Consider the specification for the type character string shown in Figure 15.[6]

CHARSTRING possesses all three of the completeness properties that have been discussed so far:

- Constructive completeness - All objects of the type are seeds, created directly by the read function.
- Testability completeness - Guaranteed as long as eof? for files is defined.
- Discriminatory completeness - Provided by equal as long as ¬ for booleans is defined.

The Data Type Character String

```
Domain:
   Source set: set of sequences of characters
   Invariant: T
Operations:
   procedure READ(var s: string; var f: file);
      pre ¬eof?(f)
      post s=first(f') ∧ f=trailer(f')
   procedure WRITE(s: string; var f: file);
      post f = s ~ f';
   function EQUAL(s1,s2: string): boolean;
      post RESULT = (s1=s2)
```

Figure 15: Abstract Specifications of the Data Type CHARSTRING

But CHARSTRING's usefulness is very limited. Why is this? The answer is that objects of the type cannot be manipulated in any way. Of course, any of a variety of operations could be added to the type to allow some kind of manipulation. For example, the following operation could be defined:

```
procedure CHOP2(var s: string);
   pre length(s) ≥ 2
   post s=trailer(trailer s'))
```

Now some manipulation is allowed, but the type is still not very useful.[7] To measure the extent to which transformations between objects can be performed, we introduce the concept of *transformational completeness*, which can be formally defined as follows:[8]

$$\forall x,y\ A(x) \wedge A(y) \wedge x \neq y \Rightarrow [\exists p_1 \ldots p_n \mid p_n[\ldots p_1(x)] = y \wedge \exists z \mid A(z) \wedge p_n[\ldots p_1(z)] \neq y]$$

In other words, there is a series of operations that can produce any object from any other. Furthermore we require that the result of the series of operations must in fact depend on the starting object (i.e., none of the p's may be constant functions). Without this requirement, transformational completeness would be implied by constructive completeness.

---

[6] This specification is also modeled on sequences.

[7] Notice that with just this addition to the specification, testability completeness is lost.

[8] We again use functional notation for convenience.

There are a variety of ways that CHARSTRING can be made transformationally complete, such as by adding concatenation and substring operations. The type rational number, as we have defined it, is transformationally complete. So are the common types stack (with CLEAR, PUSH, and POP) and set (with CLEAR, INSERT, and DELETE).

Transformational completeness is sometimes too strong a requirement. For example, suppose one wanted to define the data type MACHINE-CLOCK. Then transformational completeness is not necessary since the value of the clock need only increase; there is no circumstance under which a larger value would need to be transformed into a smaller one. Despite this, the check for transformational completeness is a good way to evaluate a new type specification since many types do require it and those that do not do so fail for specific reasons.

5.2.5 Conservation Completeness

One of the goals of completeness properties is to define a specification that contains the building blocks for a set of useful operatons even if all of those operations are not explicitly provided. So far, we have examined properties of the building blocks themselves but not looked explicitly at how easy they are to combine. Now we have to do that. Consider again the fragment of a specification for the type set, shown in Section 5.2.2. Notice that INSERT is not a pure function. It does not create a new set; instead it modifies the set that is passed to it. Sometimes this is exactly what is desired. But sometimes it is not. In particular, if INSERT is to be used as a suboperation in some new operation, it may be important that the original value of the set still be available after exectution of this new operation. For example, depending on the structure of the type of the elements of a set, it may be possible to construct a MEMBER? function from the given operations INSERT and OKTOINSERT?. But with only these operations as they are now defined, the original set will be destroyed in the process of computing MEMBER?.

We define *conservation completeness* as follows:

```
A type possess conservation completeness if it possesses the other four
completeness properties and if there exists a set of operations that,
for the specified domain, guarantees those properties and that can be
executed without side effects on objects of the type.
```

There are two ways that conservation completeness can be guaranteed:

1. Satisfy the other completeness properties with a set of operations that cause no side effects on objects of the type.

2. Guarantee that a copy (assignment) operation on the type is available so that initial values can be saved prior to executing any operation with side effects on objects of the type.

If the second option is chosen, it is not necessary that a copy operation be provided explicitly as long as one can be constructed from the operations that are provided. For example, the stack type defined in Figure 13 has conservation completeness since a stack can be copied by topping all the elements into a temporary stack and then topping them into the copy location.

Notice that we define conservation completeness only with respect to side effects on objects of the type being defined. This is necessary to allow a READ operation (such as that used in the definition of CHARSTRING shown in Figure 15) to be used to provide constructive completeness.

5.2.6 The Limitations of Completeness Properties

It is important to keep in mind that the completeness properties that have just been defined serve as additional measures of the utility of a data type specification that, for other reasons, appears to make sense. They do not, by themselves, guarantee effectiveness since many of them can be satisfied using useless operations designed for the sole purpose of passing a completeness test. For example, in Section 5.2.4, the incomplete type CHARSTRING was defined. It was then suggested that it could be made transformationally complete by adding the operations concatenate and substring. Alternatively the following operation could be substituted for substring:

```
function almostmakenull(s: string): string;
   pre T
   post if s='' then RESULT='a' else RESULT=''
```

Now to transform a string to a longer one, concatenation can be used. To transform a nonempty string to a shorter one, almostmakenull is used to create an empty string and then concatenation is used. Recall that to establish transformational completeness, it is necessary to be able to transform each object of the type into each other object without using any constant functions (since then we would not be transforming but simply creating the second object from scratch). Because almostmakenull is not quite a constant function, it is allowed in a proof of transformational completeness even though it violates the intent of the definition. But almostmakenull is a foolish function that no one would seriously consider providing.

The set of completeness properties that have been defined here is not itself "complete" in the sense that it captures all the properties necessary to make any type useful. There are other things that are important for specific classes of types. For example, it is important that, for structured types with order (e.g. arrays, lists, records), an accessing operator (such as array indexing, list next, or record field selection) be provided. What we have attempted to do here, though, is to describe a set of properties that are important for all classes of types.

References

1. Berg, H.K, W.E. Boebert, W.R. Franta, & T.G. Moher. *Formal Methods of Program Verification and Specification*. Prentice-Hall, Englewood Cliffs, N.J., 1982.

2. Dahl, O.-J., E.W. Dijkstra & C.A.R. Hoare. *Structured Programming*. Academic Press, New York, 1972.

3. Dijkstra, E.W.. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

4. Ernst, G.W. & W.F. Ogden. "Specification of Abstract Data Types in MODULA." *ACM Transactions onProgramming Languages and Systems 2*, 4 (October 1980).

5. Goguen, J.A., J.W. Thatcher & E.G. Wagner. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology, Volume IV, Data Structuring*, R. T. Yeh, Ed.,Prentice-Hall, Englewood Cliffs, N.J., 1978.

6. Gries, D. A Survey of the Use of Type. In *Programming Methodology*, D. Gries, Ed.,Springer-Verlag, New York, 1978.

7. Guttag, J.V., E. Horowitz & D.R. Musser. "Abstract Data Types and Software Validation." *Communications of the ACM 21*, 12 (December 1978).

8. Guttag, J.V. & J.J. Horning. The Algebraic Specification of Abstract Data Types. In *Programming Methodology*, D. Gries, Ed.,Springer-Verlag, New York, 1978.

9. Hoare, C.A.R. "Proof of Correctness of Data Representations." *Acta Informatica 1* (1972).

10. Liskov, B.H. & S. Zilles. "Programming with Abstract Data Types." *SIGPLAN Notices 9* (April 1974).

11. Morris, J.H., Jr. Types Are Not Sets. Proc. ACM Symposium Principles of Programming Languages, 1973.

12. Popek, G.J, J.J. Horning, B.W. Lampson, J.G. Mitchell, R.L. London. Notes on the Design of Euclid. Proc. ACM Conference on Language Design for Reliable Software, in SIGPLAN Notices, 1977.

13. Reingold, E.M., J. Neivergelt, & N. Dao. *Combinatorial Algorithms*. Prentice-Hall, Englewood Cliffs, N.J., 1977.

14. Wegbreit, B. & J.M. Spitzen. "Proving Properties of Complex Data Structures." *Journal of the ACM 23*, 2 (April 1976).

15. Wulf, W. A., R. L. London, & M. Shaw. "An Introduction to the Construction and Verification of Alphard Programs." *IEEE Trans. on Software Engineering 2*, 4 (1976).

16. Wulf, W.A, M. Shaw, P.N. Hilfinger, & L. Flon. *Fundamental Structures of Computer Science*. Addison-Wesley, Reading, Mass., 1981.