Chapter 5

Algebraic Specifications

Up to this point, we have dealt primarily with integer data types in the application programs. But to continue to do so would be very restrictive to programmers. In fact, not only do we want to find ways to reason about programs having many different primitive types (such as strings, characters, or floating point numbers), but we want to build an inference system that allows programs to be verified even when the application programmer is free to enrich the environment with new and special purpose types.

1. MORE ABOUT DATA ABSTRACTIONS

Being able to add new data types on a program by program basis is a powerful capability. This concept, which evolved in the 1970s, is known as *data abstraction* (or, looked at another way, *encapsulation*), and it is a major "divide and conquer" technique available to help programmers keep development costs down. The idea is to identify commonly used activities or information, and then, abstracting from the situation, explicitly describe that information structure in a concise and centralized way. Once the structure has been isolated, then one programmer can go off and implement the data-handling routines, another can work on the application program *using* the data, and both will be assured that the union of their efforts will function as intended.

We previous discussed data abstractions in Chapter 3. Here we extend the concept and discuss why we have this concept of "type" in the first place. Associating a "type" name with some piece of data is a declaration by the programmer concerning how that data is intended to be used. When the only thing we can declare about a datum is that it is an integer or string, then there is still much room for misuse of that information. However, when a designer abstracts some collection of desirable properties and declares this collection to be a new "type," then it is easier to infer more about the intended usage of the data. The programmer can say stronger things about the program components. In general, the way a designer will perform this abstraction activity will vary from methodology to methodology, and this is (currently) well beyond what we study in this course. However, once a new type has been defined it is essential that we be able to

1. verify that implementations of the new type preserve properties that characterize it, and
2. reason about application programs that use the new type.

For our purposes, a **data type** consists a set of values (its domain) and a set of operations on the domain. Defining the domain is usually straightforward: Small, finite domains can be enumerated. Others can be defined constructively, by first identifying a small number of base elements, and then describing how to generate the remaining elements of the domain. For instance, the $string$ domain can be constructed by saying:

- A single letter is a $string$.
- Any $string$ followed by a single letter is a $string$.
- Nothing else is a $string$.

This third assertion helps us get a bound on what can be a $string$.

Once the domain is specified, we are faced with describing the operations on it. The first step is to enumerate these operators' names, and identify in some way what types of values they both operate on (completely) and also return. This step is referred to as prescribing the *syntax*, and the way we give syntax is usually dependent on the way we intend to describe the operator's functionality (or *semantics*).

There are several ways to give semantics. One is quite informal: to give a textual description of the behavior. Due to the ambiguities of natural language, this is generally unsatisfactory. It allows ambiguities, does not lend itself to automatic checking, and requires more space to accomplish the task than other methods. Two other techniques for specifying abstract data types (ADTs) are more formal. These are *operational* specifications and *algebraic* specifications, and are described in detail in the following sections. In each case, we first concentrate on what the specifications look like (along with related issues that come up in the approach), then move on to consider how implementations of that data type are verified. We conclude discussion on each approach with detailed examples of how application programs using the ADTs are themselves verified.

2. OPERATIONAL SPECIFICATIONS

An operational specification describes an operation in terms of another type with well-defined mathematical properties, e.g., sequences, sets, or tuples. The VDL model of Chapter 1 was one general class of operational specification. As another specific example, consider specifying operations to make rational numbers from integers and to add two rational numbers using fractions as the implementation of rational numbers. Each procedure defines one of the operations using **pre** and **post** conditions that define the semantics of the operation. The postcondition is the essential operational semantics of that particular operation.

The meaning of $AddRat$ is defined using the "+" operator on fractions.

$$
\begin{array}{l}
\textbf{procedure}\ MakeRat(\textbf{var}\ Result : Rational; N, D : integer); \\
\quad \{\textbf{pre} : D \neq 0 \\
\quad \textbf{post} : Result = N/D\} \\
\textbf{procedure}\ AddRat(\textbf{var}\ Result, Op1, Op2 : Rational); \\
\quad \{\textbf{pre} : true \\
\quad \textbf{post} : Result = Op1 + Op2\}
\end{array}
$$

Specifying rational numbers using fractions does not seem like giving an implementation because there is no difference in abstraction between the two types. A better example of operational specifications is the specification of type stack in terms of type sequence[62]. For this, we need a definition for sequences and the operation which manipulate them.

$< s_1, ..., s_k >$	denotes the sequence of elements specified
$<>$	empty sequence
$s \sim x$	is the sequence resulting from concatenating element x at the end of s
$s_1 \& s_2$	is the sequence resulting from concatenating sequence s_2 after s_1
$length(s)$	the number of elements in s
$first(s)$	the leftmost element of s
$last(s)$	the rightmost element of s
$tail(s)$	the sequence resulting from deleting the first element of s
$lead(s)$	the sequence resulting from deleting the last element of s
$seq(V, n, m)$	$< V[n], V[n+1], ..., V[m] >$; also $seq(V, n, m) = seq(V, n, m-1) \sim V[m]$

Note that first, last, tail, and lead are undefined for $<>$.

Assuming we have this notation for sequences, we define type stack as follows.

$$
\begin{array}{l}
\textbf{const } MaxDepth := \ldots \\
\textbf{type } Stack := \ldots \\
\{\textbf{invariant :} 0 \leq length(S) \leq MaxDepth\} \\
\textbf{procedure } NewStack(\textbf{var} S : Stack); \\
\quad \{\textbf{pre} : true \\
\quad \textbf{post} : S = <>\} \\
\textbf{procedure } Push(\textbf{var} S : Stack; E : StackElt); \\
\quad \{\textbf{pre} : length(S) < MaxDepth \\
\quad \textbf{post} : S = S' \sim E\} \\
\textbf{procedure } Pop(\textbf{var} S : Stack); \\
\quad \{\textbf{pre} : true \\
\quad \textbf{post} : (length(S') = 0 \wedge S = S') \vee (length(S') > 0 \wedge S = lead(S')\} \\
\textbf{function } Top(\textbf{var} S : Stack) : StackElt; \\
\quad \{\textbf{pre} : length(S) > 0 \\
\quad \textbf{post} : Top = last(S)\} \\
\textbf{function } Empty(\textbf{var} S : Stack) : boolean; \\
\quad \{\textbf{pre} : true \\
\quad \textbf{post} : Empty = (length(S) = 0)\}
\end{array}
$$

These specifications are relatively easy to construct because they seem like programming, although they have been criticized since they suggest implementations too strongly. Thus, one might argue that operational specifications should be very inefficient implementations so that no one is tempted to use the specification as an implementation.

3. ALGEBRAIC SPECIFICATION OF ADTS

An alternative to operational specification of ADTs is to define them *algebraically*. In this approach, we define the semantics of ADT operators in terms of how they *inter*act with one another, instead of how they act in terms of a concrete type. It may be a bit harder to come up with good axioms in algebraic specification as opposed to operational specifications — after all, in operational specifications we deal with implementations of single operations.

Algebraic specifications are divided into two parts: syntax and semantics. The syntactic description is often referred to as the *signature* of the algebra, and each "auxiliary" data type used to give this signature is called a *sort*. For instance, the sorts for STACK above are ***integers*** and ***booleans***.

$$\begin{array}{ll} push: & STACK \times INTEGERS \rightarrow STACK \\ pop: & STACK \rightarrow STACK \\ top: & STACK \rightarrow INTEGERS \cup \{undefined\} \\ empty: & STACK \rightarrow BOOLEAN \\ newstack: & \rightarrow STACK \end{array}$$

The domain of STACK (as opposed to the domain of any of the above operators) may now be described. This can actually be more difficult than it sounds. Informally, we know that we want our type STACK to encompass all the stacks of integers that could arise in some program. Formally, we need a way to describe them. Since we have have listed the operators, we can now identify a subset of the operators as being *constructors* for the type. That is, we want to identify the least number of operators necessary to be able to construct all possible stacks that could come up. Intuitively, we can believe that ***newstack*** and ***push*** are constructors, since any stack that was formed by using a ***pop*** somewhere along the way can be formed by using only a smaller number of calls to ***push***.

Axioms describe the meanings of operators in terms of how they interact with one another. For instance, our ADT STACK can be specified as:

$$\begin{array}{lll} 1. & pop(newstack) = & newstack \\ 2. & pop(push(S, I)) = & S \\ 3. & top(newstack) = & undefined \\ 4. & top(push(S, I)) = & I \\ 5. & empty(newstack) = & true \\ 6. & empty(push(S, I)) = & false \end{array}$$

where ***S*** and ***I*** are considered to be universally quantified as an instance of stack and integer, respectively. Consider Axiom 1 above: Remember we are not declaring that there is some special instance of a stack that is distinguished by being a "newstack." Instead, this axiom is read "the effect of composing a call to ***pop*** and a call to my function ***newstack*** is just the same as having called ***newstack*** directly." Likewise, Axiom 2 is read "the effect of composing a our ***pop*** function with a call to ***push*** on some stack ***S*** and an integer ***I*** yields a stack which is just the same as the stack ***S*** we started with."

This choice of axioms defines an abstract type that we call STACK, and seems to capture the basic semantics of the stack of our intuition. Note what this specification does *not* capture, for instance, boundedness. The above specification allows us to generate arbitrarily deep stacks, which might not correspond to what our underlying implementations should be expected to support. To capture boundedness, we might also define a "hidden function" (i.e. an operator not normally exported

for users to access, but which allows us to capture a desired property in our specification) such as ***size***:***STACK***→INTEGERS, then use the following algebra in place of our earlier specification:

1. $pop(newstack) =$ $\quad newstack$
2. $pop(push(S, I)) =$ $\quad if\ size(S) = MAX\ then\ pop(S)\ else\ S$
3. $top(newstack) =$ $\quad undefined$
4. $top(push(S, I)) =$ $\quad if\ size(S) = MAX\ then\ top(S)\ else\ I$
5. $empty(newstack) =$ $\quad true$
6. $empty(push(S, I)) =$ $\quad false$
7. $size(newstack) =$ $\quad 0$
8. $size(push(S, I)) =$ $\quad if\ size(S) = MAX\ then\ size(S)$
   $else\ size(S) + 1$

where MAX is some agreed-upon parameter.

In general, a list of axioms appears as a set of equations, with each left-hand side containing a composition of operators, and each right-hand side containing a description of how the composition behaves in terms of the type's operators and simple "if-then-else" constructs. Iteration or other auxiliary variables are not used.

These axioms are often thought of as "rewrite rules." Informally, this is because we can take one instance of the ADT and appeal to the list to find a simpler way to express that instance. However, there is a more mathematical criteria for rewrite rules, which will be explored later. In the meantime, the key is that all axioms should take one composition of functions and show how it can be expressed in only one of the arguments or functions, that is, we rewrite the composition in a simpler way.

Notice also that this functional approach to building ADTs treats instances of the type as being "immutable." (In contrast, and from an implementation point of view, we usually think of a stack as being one object which accumulates side effects based on the operations on the stack.) According to the specification above, all the ***push*** operation guarantees is that it returns an instance of STACK which is constructed from another instance of STACK and some INTEGER. This situation, together with our ability to view the axioms as rewrite rules, allows us to think of instances of an algebraically specified ADT as being sentences in a language, a very useful perspective. For example, one sentence might be formed by calling ***newstack***, then applying a ***push***, then another ***push***, then a ***pop***, then applying the query ***empty***, for example,

$$empty(pop(push(push(newstack, 42), 17)))$$

Using Axiom 2 above (from unbounded stacks), we can simplify the outermost composition of ***pop*** and ***push***, and rewrite the sentence as:

$$empty(push(newstack, 42))$$

At this point we can appeal to axiom (6) and simplify the string to be simply ***false***. Note how we were able to squeeze the call to ***pop*** out, and express the argument to ***empty*** using a sentence which only contains calls to either ***push*** or ***newstack***. This illustrates our point in the earlier discussion concerning constructors. We will often refer to an instance of the ADT which is built only through use of constructor operations as being in *normal* (or *canonical*) *form*. In general, it is very useful to look at intermediate expressions as being strings in a language to which transformations can be applied. This will assist us in our reasoning about programs using algebraically specified ADTs.

3.1. Developing Algebraic Axioms

For the axiomatization of type T to be sufficiently complete, it must assign a value to each term of type T [24]. Since the constructors produce all the values of the domain of the type being defined, then we need only specify how non-constructor operations affect each constructor. Thus if $c1_1$ and c_2 are constructors operations of type T and f is a non-constructor operation whose signature is $r: T \rightarrow T1$, we write axioms to define $f(c_1)$ and $f(c_2)$. For example, for type stack $push$ and $newstack$ are constructors, while pop, top and $empty$ are not constructors. Thus, we wrote axioms to define the meanings of:

$$
\begin{array}{l}
pop(newstack) \\
pop(push(S,i)) \\
top(newstack) \\
top(push(S,i)) \\
empty(newstack) \\
empty(push(S,i))
\end{array}
$$

Consider another pair of examples. Let us define the ADT "SET of integers," having syntax

$$
\begin{array}{ll}
newset: & \rightarrow SET \\
insert: & SET \times INTEGER \rightarrow SET \\
delete: & SET \times INTEGER \rightarrow SET \\
member: & SET \times INTEGER \rightarrow BOOLEAN
\end{array}
$$

having semantics

$$
\begin{array}{ll}
member(newset,i) = & false \\
member(insert(S,i),j) = & if\ i = j\ then\ true\ else\ member(S,j) \\
delete(newset,i) = & newset \\
delete(insert(S,i),j) = & if\ i = j\ then\ delete(S,j) \\
 & else\ insert(delete(S,j),i)
\end{array}
$$

Alternately, we can define the ADT "BAG of integers" as:

$$
\begin{array}{ll}
newbag: & \rightarrow BAG \\
insert: & BAG \times INTEGER \rightarrow BAG \\
delete: & BAG \times INTEGER \rightarrow BAG \\
member: & BAG \times INTEGER \rightarrow BOOLEAN
\end{array}
$$

having semantics

$$
\begin{array}{ll}
member(newbag,i) = & false \\
member(insert(B,i),j) = & if\ i = j\ then\ true\ else\ member(S,j) \\
delete(newbag,i) = & newbag \\
delete(insert(B,i),j) = & if\ i = j\ then\ B \\
 & else\ insert(delete(B,j),i))
\end{array}
$$

It would be very useful for you to go through a few simple examples using these two close but different sets of axioms. Build a few sentences in the "language" of the type, and verify for yourself that, for instance, you can add several copies of the same integer to a set, and that after removing it you will be able to ask "member" of that set and integer and get the right answer. In the next subsections, we give a more complete development of a set of axioms.

We have more choices for axioms when defining operations which have more than one argument of the type being defined. The examples below illustrate several different sets of axioms defining add for the type natural numbers generated by 0 and $succ$. Each of the sets of axioms are sufficiently complete.

$$
\begin{array}{ll}
add(0, X) = & X \\
add(succ(Y), Z) = & succ(add(Y, Z))
\end{array}
$$

$$
\begin{array}{ll}
add(0, 0) = & 0 \\
add(succ(X), 0) = & succ(X) \\
add(Y, succ(Z)) = & succ(add(Y, Z))
\end{array}
$$

$$
\begin{array}{ll}
add(0, 0) = & 0 \\
add(0, succ(W)) = & succ(W) \\
add(succ(X), 0) = & succ(X) \\
add(succ(Y), succ(Z)) = & succ(succ(add(Y, Z)))
\end{array}
$$

Huet and Hullot describe an algorithm for determining if the set of arguments of an operation's axioms is sufficiently complete. [33] For each operation, a set of n-tuples representing the operation's arguments is constructed. Initially all the arguments in the first positions of the axioms for a particular operation are examined to determine that they "cover" the set of constructors, either by containing an instance of each constructor or by containing a variable which can represent any constructor. For example, the second set of axioms defining add have the following set of argument two-tuples: $\{ < 0,\ 0 >, < succ(X),\ 0) >, < Y,\ succ(Z) > \}$. We consider the arguments in the first positions: $0,\ succ(X),$ and Y, and conclude that all constructors are present and that $succ$'s argument covers all its constructors because it is a variable. If all the constructors are present, we consider each set of n-1-tuples formed from that subset of the axioms having the first constructor or a variable in their first positions, followed by the set of n-1-tuples formed from the subset of axioms having the second constructor or a variable in their first positions, etc. until we have processed all the constructors. Again, by example, we consider the sets of 1-tuples $\{ < 0 >, < succ(Z) > \}$ (whose first arguments were 0 or Y) and $\{ < 0 >, < succ(Z) > \}$ (whose first arguments were $succ(X)$ or Y). Since all the constructors are present in each of these sets and $succ$'s argument covers all its constructors, the axioms are sufficiently complete.

Huet and Hullot's algorithm is given below.

```
function complete(F: {f1,...,fm} where fi = < si^1, ..., si^k >): boolean;
  for every fi do if fi contains duplicate variables then
    return(false);
  if k = 0 and F = { <> } then return(true);
  for i in 1..m do
    if si^1 is a variable and complete({< si^2, ..., si^k >| si^1
       is a variable} ) then return(true);
  for every constructor c(p1,...,pn) do
    if there exists one si^1 with leading function symbol c and
    complete({< p1, ..., pn, si^2, ..., si^k >| si^1 = c(p1,...,pn)}
            ∪ {< X1, ..., Xn, si^2, ..., si^k >|
                 si^1 is a variable and Xj's are fresh variables})
        then skip
        else return(false);
  return(true)
```

3.2. Hints For writing algebraic axioms

The process of writing algebraic specifications is similar to that of programming.[1] The main difference is that in programming we are interested in the efficiency of the solutions, whereas in specifying we are interested in expressing the functionality and nothing else (i.e., avoid implementation bias). This means that in writing specifications we want to express only the function to be computed, not the method of computation. Besides, algebraic specification languages do not have the concept of side effect (e.g., variable assignment) every time a new value is created, so algebraic specifications resemble so-called functional programs.

Determine functionality

You should determine the use of the data type, what operations create instances of the data type, what operations modify instances,[2] what operations obtain values from the data type, etc. To determine these functions you must think about the uses of the data type. This might be hard, especially if this is a "general-purpose" type and you cannot anticipate all its uses.

Write signatures

Once you have decided which is the set of functions, you write the signatures of them. In this step you can provide a lot of insight on the abstract type if you use appropriate, meaningful names. Note, however that the final word on meaning is not the name: *This is the whole point of using formal specifications.*

Choose constructors

Initially all functions that yield the defined type are candidates for constructors. In principle, you can have them all as constructors, but that will introduce many problems both in the consistency of your axioms and also in the number of axioms (equations) that you have to provide. Hence you want

[1] We are indebted to Pablo Straub of the Catholic University of Chile for contributing this part.

[2] In the specification no modification is allowed, but modification is expressed by creation of a new instance that is very similar to the original.

a "minimal" set of constructors. This set of constructors will have at least one function that has no parameter of the type defined (usually it will be a function with no parameters at all).

If you can figure out how to express the semantics of a function in terms of others, then this function is not a constructor. Besides, the way you express the semantics of the function will dictate the axioms you will write later.

Sometimes which operations are constructors is evident from the specification; sometimes there are explicit choices to be made. For example, consider a specification for a dequeue, a list with insertion and deletion on both ends. We have two possible sets of constructors: {AddFront, New} and {AddBack, New}. Which one to use is a matter of taste, but you should decide on one of them.

Write axioms

The first part is to determine which axioms you need to write. Constructors do not need axioms: you will only write axioms for nonconstructors. For each nonconstructor function, you need to cover all cases of the parameters. As a rule of thumb, if there are n constructors you will need n equations for each nonconstructor.

This rule of thumb is not complete, though, because sometimes there is more than one parameter of the defined type, so you may need more axioms. For example if you are defining a function with 3 parameters of the type defined and you have 2 constructors, you may need $2^3 = 8$ equations to cover all constructor combinations. Besides, sometimes you need to consider nesting of constructors to discriminate different cases. Fortunately, this rarely occurs in practice, and you will usually be able to cover all cases with just a few equations.

Now that you know all cases to consider, you write the left-hand side of all axioms. There are no general methods to write the right-hand side, though. While writing the axioms you will notice that some left hand sides are meaningless (e.g., extract information from an empty data structure). From this you define your exceptions.

Example: Finite integer functions

We want to design a data structure to represent finite integer functions. A finite function is a function whose domain is finite, that is, the function is defined in a finite number of points. Finite functions can be represented by a lookup table whose entries are ordered pairs. In this case these ordered pairs will be pairs of integers.

Note that even though here we are defining the data structure for the case of integers, the specification that we will develop can be parameterized for finite functions with any domain and range.

Determine functionality

We want to be able to manipulate finite mappings by creating simple mappings, modify mappings at a specific point (define the value, make the value undefined), and make the union of mappings. The union of two mappings is not commutative, because if both mapping are defined at a particular point, the definition from the second mapping takes precedence. We need to be able to handle empty mappings and also have a function to create a single-entry mapping. Obviously, given an integer, we want to know the value of the function at that point. Besides, given a mapping, we want to get its domain.

Writing signatures

Given the requirements expressed before, define the following functions with their corresponding signatures.

EmptyMapping:		$\rightarrow$ mapping
SingleMapping:	integer $\times$ integer	$\rightarrow$ mapping
Define:	mapping $\times$ integer $\times$ integer	$\rightarrow$ mapping
Undefine:	mapping $\times$ integer	$\rightarrow$ mapping
Union:	mapping $\times$ mapping	$\rightarrow$ mapping
Apply:	mapping $\times$ integer	$\rightarrow$ integer
Domain:	mapping	$\rightarrow$ set[integer]

Choosing constructors

Candidates for constructors are:

$$EmptyMapping, SingleMapping, Define, Undefine, Union$$

Of them, either $EmptyMapping$ or $SingleMapping$ must be chosen as constructor, because these are the only ones that do not have a parameter of type mapping. $SingleMapping$ can be easily expressed by defining an $EmptyMapping$. Further thought can lead us to choose $EmptyMapping$ and $Define$ as the constructors.

Writing axioms

First, consider which axioms we need. Our rule of thumb tells us that we need two axioms for each nonconstructor, with the caveat that $Union$ might require four axioms. Besides, since $SingleMapping$ has no parameters of type mapping, we can define it with only one axiom.

The left-hand sides are then:

$$SingleMapping(i, v) =$$

$$Undefine(Empty, i) =$$
$$Undefine(Define(m, i_1, v), i_2) =$$

$$Apply(Empty, i) =$$
$$Apply(Define(m, i_1, v), i_2) =$$

$$Domain(Empty) =$$
$$Domain(Define(m, i, v)) =$$

We delay the left-hand sides for the Union operation because we do not know yet whether we will be able to do it with two or four equations.

Now we define the semantics of all operations except Union.

$$
\begin{array}{ll}
SingleMapping(i, v) = & Define(Empty, i, v) \\
Undefine(Empty, i) = & Empty \\
Undefine(Define(m, i_1, v), i_2) = & if\ i_1 = i_2\ then\ Undefine(m, i_2) \\
 & else\ Define(Undefine(m, i_2), i_1, v) \\
Apply(Empty, i) = & undefined \\
Apply(Define(m, i_1, v), i_2) = & if\ i_1 = i_2\ then\ v\ else\ Apply(m, i_2) \\
Domain(Empty) = & NullSet \\
Domain(Define(m, i, v)) = & AddMember(Domain(m), i)
\end{array}
$$

Now let us consider function $Union$. Obviously union with the $Empty$ mapping does not modify the mapping. So as long as one of the mappings is empty the result is trivial. What happens with two nonempty mappings? We want the value of the second mapping to take precedence. Here we present 4 equations to consider all possible cases.

$$
\begin{array}{l}
Union(Empty, Empty) = Empty \\
Union(Define(m, i, v), Empty) = Define(m, i, v) \\
Union(Empty, Define(m, i, v)) = Define(m, i, v) \\
Union(Define(m_1, i_1, v_1), Define(m_2, i_2, v_2)) = \\
\quad Define(Union(Define(m_1, i_1, v_1), m_2), i_2, v_2)
\end{array}
$$

By looking at the complex specification of the $Union$ function we note that we can use only two equations, because the first parameter is never used in recursion: It is always taken *in toto*. If the second parameter is $Empty$ we always return the first; if the second parameter is not $Empty$ we add all of its definitions to the first one.

The new equations are these

$$
\begin{array}{ll}
Union(m, Empty) = & m \\
Union(m_1, Define(m_2, i, v)) = & Define(Union(m_1, m_2), i, v)
\end{array}
$$

3.3. Consistency

Axioms sets are *consistent* if we always get the same final result, independent of the order in which we apply the axioms to simplify terms. As an example (adapted from [27]), consider adding the following axiom to the definition of BAGs above:

$$member(delete(S, i), j) = if\ i = j\ then\ false\ else\ member(S, j)$$

Then consider the simplification of string:

$$member(delete(insert(insert(newbag, 3), 3), 3), 3)$$

By first applying our new axiom, this can be rewritten as

$$if\ 3 = 3\ then\ false\ else\ member(\ldots)$$

or ***false***. However, by our original ***delete(insert())*** axiom, this can become

$$member(if\ 3 = 3\ then\ insert(newbag, 3)\ else\ delete(\ldots)$$

or

$$member(insert(newbag, 3), 3)$$

or

$$if\ 3 = 3\ then\ true\ else\ member(\ldots)$$

or ***true***, a contradiction. In general, questions of consistency can be answered by using the Knuth–Bendix algorithm [38], which is described in detail later in this chapter.

3.4. Term Equality

Other interesting predicaments arise. Consider two strings for SETs,

insert(insert(newset, 3), 4)

and

insert(insert(newset, 4), 3)

Intuitively we want the sets represented by these two strings to be equal. After all, we see each as being a set of two elements, 3 and 4, and we know a property of sets is that there is no order property. However, it is impossible to prove this using only the axioms given to define the SET semantics.

In general, questions of equality are very difficult to deal with. In so-called "initial algebras," two sentences are considered equal only if they can be proven so as a consequence of axioms. Some researchers, such as Guttag [17], say that two sentences are equal *unless* one of the operations mapping out of the type of interest carries them to distinct values. In effect, the question of equality is then punted into the equality interpretation of the ADT's sorts. Regardless, when you discuss equality in the manipulation of abstract data types, it is essential to be absolutely clear as to "which equals" you are talking about.

4. DATA TYPE INDUCTION

Data type induction is an important technique in verifying properties of algebraic specifications. This induction works as follows, and is closely related to natural induction of the positive integers.

Given the set of operations defined on some ADT X, we can separate them into three sets:

1. $\{f_i\}$ are those functions which return objects of type X but do not contain arguments of type X. These generating functions create the primitive objects of type X.

2. $\{g_i\}$ are those functions which return objects of type X and do contain an argument of type X. These generating function creates non-primitive objects of type X.

3. $\{h_i\}$ are all other functions, generally taking arguments of type X and returning values of some other type. These represent uses of the type.

For the stack ADT, we get:

$$\begin{aligned} f = &\ \{newstack\} \\ g = &\ \{push, pop\} \\ h = &\ \{empty, top, size\} \end{aligned}$$

In general there will be one f member, one or two g members, and a small number of h members, but there are no prohibitions on more than these.

Let $P(x)$ be some predicate concerning $x \in X$ for ADT X. Under what conditions will P be true for every member of type X? Similar to natural induction, we want to show that primitive objects are true under predicate P and that we can use these primitive values to show that by constructing new values of the type, the property remains true.

We can define *data type induction* as follows:

- Given ADT X with functions f_i, g_i and h_i defined as above and predicate $P(x)$ for $x \in X$.
- Show that $P(f_i)$ is valid (base case).
- Assume $P(x)$ is valid. Show that this implies $P(g_i(x))$ is valid (i.e., $P(x) \vdash P(g_i(x))$).
- We can then conclude $\forall x \in X\ P(x)$.

For the stack ADT, this would mean that we show $P(newstack)$ and $P(x) \vdash P(g_i(x))$ (i.e., both $P(push(x, i))$ and $P(pop(x))$.) However, we do not need to show this for pop since pop is not needed to generate all stacks. The *normal-form lemma* gives us this. This lemma is a formal assertion of what we have already informally claimed earlier, namely that all instances of a STACK can be built using only calls to $newstack$ and $push$; that is, each stack has a normal form.

Lemma: For every $S \in$ STACK, either

1. $S =$ ***newstack***, or
2. $\exists S_1 \in$ STACK and $I \in$ INTEGER such that $S = push(S_1, I)$

(The uniqueness of this normal form requires a separate proof, and is left as an exercise.).

This lemma simply says that we can replace any pop operation by an equivalent set of $newstack$ or $push$ operations. The proof of this lemma is quite complex and beyond the scope of this book. It depends upon the following argument:

1. The STACK ADT is primitive recursive and each object is constructible by a finite application of the constructors.

2. "Count" the number of function applications in constructing stack S.

3. For each instance of $pop(S_i)$, consider the definition of S_i. If it is of the form $newstack$, then $pop(newstack) = newstack$. If it is of the form $pop(push(S_j, j))$, then by axiom, this is just S_j. If it is of the form $pop(pop(S_j))$ apply the same argument to S_j until you get a $pop(newstack)$ or $pop(push(\ldots))$.

4. In all of the above cases, the "count" of function applications by applying these reductions will be less. Thus, the process must terminate.

5. This can be repeated as long as a pop operation remains. So all pop operations can be removed from any stack description.

This explanation was generally intuitive. See the Guttag and Horning reference [24] at the end of the chapter for complete details on the proof of the normal form lemma.

This normal form lemma is not essential for verifying properties of an ADT. However, proving such a lemma is often a way to reduce the amount of work associated with verifications because arguments which previously needed to be made for ***all*** operators returning the type of interest can then be replaced by an argument for only the ***constructors*** of the type.

4.1. Example: Data Type Induction Proof

Using data type induction, we can prove that pushing a value on a stack increases its size by proving the theorem $size(push(S, X)) > size(S)$. The axioms for stacks are shown below.

$$
\begin{array}{ll}
axiom\ top_1: & top(newstack) = undefined \\
axiom\ top_2: & top(push(S, I)) = I \\
axiom\ size_1: & size(newstack) = 0 \\
axiom\ size_2: & size(push(S, I)) = size(S) + 1 \\
axiom\ pop_1: & pop(newstack) = newstack \\
axiom\ pop_2: & pop(push(S, I)) = S
\end{array}
$$

We start the proof of $size(push(S, X)) > size(S)$ by using the axioms until no more simplifications are possible.

$$
\begin{array}{ll}
size(push(S, X)) > size(S) & \\
size(S) + 1 > size(S) & by\ size_2
\end{array}
$$

At this point, we resort to data type induction. First we replace S by $newstack$ and show the theorem holds.

$$
\begin{array}{ll}
Begin\ induction\ on\ S = newstack & \\
size(newstack) + 1 > size(newstack) & replace\ S\ with\ newstack \\
0 + 1 > 0 & by\ size_1 \\
true &
\end{array}
$$

Next, we assume that the theorem holds for S', i.e, $size(S') + 1 > size(S')$, and show the theorem holds for $S = push(S', X)$.

$$
\begin{array}{ll}
Induction\ on\ S = push(S', X) & \\
Inductive\ hyp.: size(S') + 1 > size(S') & \\
size(push(S', X)) + 1 > size(push(S', X)) & push(S', X)\ replaces\ S \\
size(S') + 1 + 1 > size(S') + 1 & by\ size_2 \\
size(S') + 1 > size(S') & subtract\ 1\ from\ each\ side \\
true & by\ inductive\ hypothesis
\end{array}
$$

5. VERIFYING ADT IMPLEMENTATIONS

5.1. Verifying Operational Specifications

Proving that operations are implemented correctly is difficult because of the abstraction gap between the objects in the specifications (sequences) and those in the implementation (arrays and integers). We add two new pieces of documentation to bridge this gap: a representation mapping (R) that maps implementation objects to specification objects, and implementation-level input and output assertions for each operation. For the stack example, these are:

$$
\begin{array}{l}
\textbf{const}\ MaxDepth := \ldots \\
\textbf{type}\ Stack := \textbf{record}\ Sp : 0..MaxDepth; \\
\quad V : \textbf{array}[1..MaxDepth]\ \textbf{of}\ \ldots \textbf{end} \\
\{\textbf{invariant} : 0 \leq S.Sp \leq MaxDepth\} \\
\{\textbf{representation mapping} : R(S.V, S.Sp) = seq(S.V, 1, S.Sp)\} \\
\textbf{procedure}\ NewStack(\textbf{var} S : Stack); \\
\quad \{\textbf{in} : true \\
\quad \textbf{out} : S.Sp = 0\} \\
\textbf{procedure}\ Push(\textbf{var} S : Stack; E : StackElt); \\
\quad \{\textbf{in} : S.Sp < MaxDepth \\
\quad \textbf{out} : S.V = alpha(S'.V; S'.Sp + 1 : E) \wedge S.Sp = S'.Sp + 1\} \\
\textbf{procedure}\ Pop(\textbf{var} S : Stack); \\
\quad \{\textbf{in} : true \\
\quad \textbf{out} : (S'.Sp = 0 \wedge S.Sp = S'.Sp) \vee \\
\qquad (S'.Sp > 0 \wedge S.Sp = S'.Sp - 1)\} \\
\textbf{function} Top(\textbf{var} S : Stack) : StackElt; \\
\quad \{\textbf{in} : S.Sp > 0 \\
\quad \textbf{out} : Top = S.V[S.Sp]\} \\
\textbf{function} Empty(\textbf{var} S : Stack) : boolean; \\
\quad \{\textbf{in} : true \\
\quad \textbf{out} Empty = (S.Sp = 0)\}
\end{array}
$$

The verification steps are:

• Prove that any initialized concrete object is a well-formed abstract object, that is, if any implementation object X satisfies the concrete invariant, its corresponding abstract value (formed by applying the representation mapping) satisfies the abstract invariant.
$CI(X) \Rightarrow AI(R(X))$

• For each operation, show that its implementation is consistent with its in and out conditions:

$$in(X) \wedge CI(X)\{operation\ body\}out(X) \wedge CI(X)$$

This proof is identical to those presented in Chapter 2. However, if $post(X)$ contains ghost variables, we may assertions to $in(X)$ equating input values with their respective ghost variables.

• For each operation, show its abstract precondition guarantees that its concrete precondition is true, and that its concrete postcondition guarantees that its abstract postcondition is true:

$$
\begin{array}{ll}
a. & CI(X) \wedge pre(R(X)) \Rightarrow in(X) \\
b. & CI(X) \wedge pre(R(X')) \wedge out(X) \Rightarrow post(R(X))
\end{array}
$$

Example verifications

1. $CI(X) \Rightarrow AI(R(X))$

$$0 \leq S.Sp \leq MaxDepth \Rightarrow 0 \leq length(seq(S.V, 1, S.Sp)) \leq MaxDepth$$
$$0 \leq S.Sp \leq MaxDepth \Rightarrow 0 \leq S.Sp \leq MaxDepth$$

2. $\{in(X) \wedge CI(X)\}$ Push's body $\{out(X) \wedge C(X)\}$

procedure $Push($**var**$S : Stack; E : StackElt);$
  $\{$**in** $:S.Sp < MaxDepth$
  out $:S.V = alpha(S'.V; S'.Sp + 1 : E) \wedge S.Sp = S'.Sp + 1\}$
  begin
    $S.Sp := S.Sp + 1;$
    $S.V[S.Sp] := E$
  end

Since the post-condition contains ghost variables $S'.V$ and $S'.Sp$, we add assertions to the pre-condition equating $S.V$ and $S'.V$, and $S.Sp$ and $S'.Sp$ to obtain the pre-condition:

$$\textbf{in} : S.V = S'.V \; land \; S.Sp = S'.Sp \; \wedge \; S.Sp < MaxDepth$$

First we use the array assignment axiom:

$$\begin{array}{rll} Q \stackrel{\triangle}{=} & \{\alpha(S.V; S.Sp : X) = \alpha(S'.V; S'.Sp + 1 : X) \\ & \wedge S.Sp = S'.Sp + 1)\}S.V[S.Sp] := E \\ post \stackrel{\triangle}{=} & \{(S.V = \alpha(S'.V; S'.Sp + 1 : E) \wedge S.Sp = S'.Sp + 1)\} \end{array}$$

Next, we use the assignment axiom:

$$\begin{array}{rll} P \stackrel{\triangle}{=} & \{\alpha(S.V; S.Sp + 1 : X) = \alpha(S'.V; S'.Sp + 1 : X) \\ & \quad \wedge S.Sp + 1 = S'.Sp + 1)\}S.Sp := S.Sp + 1 \\ Q \stackrel{\triangle}{=} & \{\alpha(S.V; S.Sp : X) = \alpha(S'.V; S'.Sp + 1 : X) \\ & \wedge S.Sp = S'.Sp + 1)\} \end{array}$$

By the rule of composition:

$$\frac{\{P\}S.Sp := S.Sp + 1\}Q, Q\{S.V[S.Sp] := E\{post\}}{\{P\}S.Sp := S.Sp + 1; S.V[S.Sp] := E\{post\}}$$

Finally, we invoke the rule of consequence:

$$\frac{pre \Rightarrow P, \{P\}S.Sp := S.Sp + 1; S.V[S.Sp] := E\{post\}}{\{pre\}S.Sp := S.Sp + 1; S.V[S.Sp] := E\{post\}}$$

3a. $CI(X) \wedge pre(R(X)) \Rightarrow in(X)$

$$0 \leq S.Sp \leq MaxDepth \wedge length(seq(S.V, 1, S.Sp)) < MaxDepth \Rightarrow \\ S.Sp < MaxDepth$$

$$0 \leq S.Sp \leq MaxDepth \wedge length(seq(S.V, 1, S.Sp)) < MaxDepth \\ \Rightarrow S.Sp < MaxDepth$$

Rewriting $length$ by its definition yields:

$$0 \leq S.Sp \leq MaxDepth \wedge S.Sp < MaxDepth \Rightarrow \\ S.Sp < MaxDepth$$

3b. $CI(X) \wedge pre(R(X')) \wedge in(X) \Rightarrow post(R(X))$

$$0 \leq S.Sp \leq MaxDepth \wedge length(seq(S'.V, 1, S'.Sp)) < MaxDepth \wedge \\ S.V = \alpha(S'.V; S'.Sp + 1 : E) \wedge S.Sp = S'.Sp + 1 \Rightarrow \\ seq(S.V, 1, S.Sp) = seq(S'.V, 1, S'.Sp) \sim E$$

Rewriting length by its definition yields:

$$0 \leq S.Sp \leq MaxDepth \wedge S'.Sp < MaxDepth \wedge \\ S.V = \alpha(S'.V; S'.Sp + 1 : E) \wedge S.Sp = S'.Sp + 1 \Rightarrow \\ seq(S.V, 1, S.Sp) = seq(S'.V, 1, S'.Sp) \sim E$$

Replacing S.V and S.Sp on the right side of the equation yields:

$$0 \leq S.Sp \leq MaxDepth \wedge S'.Sp < MaxDepth \wedge \\ S.V = \alpha(S'.V; S'.Sp + 1 : E) \wedge S.Sp = S'.Sp + 1 \Rightarrow \\ seq(\alpha(S'.V; S'.Sp + 1 : E), 1, S'.Sp + 1) = seq(S'.V, 1, S'.Sp) \sim E$$

From the definition of sequence, $seq(V, n, m) = seq(V, n, m-1) \sim V[m]$. Thus

$$seq(\alpha(S'.V; S'.Sp + 1 : E), 1, S'.Sp + 1) = \\ seq(\alpha(S'.V; S'.Sp + 1 : E), 1, S'.Sp) \sim \\ \alpha(S'.V; S'.Sp + 1 : E)[S'.Sp + 1]$$

By the definition of the α notation:

$$\alpha(S'.V; S'.Sp + 1 : E)[S'.Sp + 1] = E \\ seq(\alpha(S'.V; S'.Sp + 1 : E), 1, S'.Sp) = seq(S'.V, 1, S'.Sp)$$

Thus the previous formula can be written as:

$$seq(\alpha(S'.V; S'.Sp + 1 : E), 1, S'.Sp + 1) = seq(S'.V, 1, S'.Sp) \sim E$$

Finally, the right side of the formula reduces to $true$.

$$0 \leq S.Sp \leq MaxDepth \wedge S'.Sp < MaxDepth \wedge$$
$$S.V = \alpha(S'.V; S'.Sp + 1 : E) \wedge S.Sp = S'.Sp + 1 \Rightarrow$$
$$seq(S'.V, 1, S'.Sp) \sim E = seq(S'.V, 1, S'.Sp) \sim E$$

5.2. Verifying Algebraic Specifications

In operationally specified ADTs, the actions of operators were described in terms of a distinct, concrete type; hence, their implementation is easily discussed in terms of only that type. In contrast, the operators of an algebraically specified ADT are described by how they interrelate with the other operators. There is no second (let alone concrete) type used in this case. Therefore, to give an implementation of an algebraically specified ADT, it is essential that we first describe the properties of a second data type that will be used to implement the first. In keeping with the spirit of this abstraction activity, we usually give an algebraic specification for this second type as well.

We often refer to the type we are trying to implement as a "higher level" data type, and to the type used in the implementation as a "lower level" data type. However, these phrases are vague, and they only result from the way we naturally think about the implementation process. In fact, the lower level type might itself be an elaborate ADT with rich semantics. It, in turn, might be implemented in terms of yet a lower-level type, and so on. A hierarchy of data types can often result in this manner. The goal of course is for one of these lower level types to ultimately be directly representable and implemented in hardware (for instance, we will shortly show how to implement the STACKs specified earlier in terms of an algebraically specified array, and we would expect that this array could be directly implemented in our host language on most computers).

However, there is one glaring problem associated with this type hierarchy, and that is how to check whether a lower-level data type is faithfully implemented on a real machine. In the example we are about to give (STACKs in terms of arrays), our specification of array assumes that we have an unbounded array, and further, that uninitialized entries in the array can be detected. These properties are rarely true of machine implementations. Hence, our techniques are only as reliable as is our implementation of the base types. Nonetheless, if we do develop a specification which can be reliably implemented on a machine, then all the benefits of a verified implementation can be enjoyed. One of the most important of these benefits is that the machine-specific properties that must be captured are very *localized*, that is, developers will be free to use instances of our STACK data type, and it will be transparent whether they are implemented using arrays (on some hardware) or actual stacks (on a stack machine).

Once we have the specification for both the higher- and lower-level types, we can describe how the one is to be implemented in terms of the other. This is done through an *implementation program*, which, just as with the original specification of the individual ADTs, looks like a list of axioms. However, whereas before the axioms showed how the operators from any one type lnterrelated, the implementation program shows how the higher-level type behaves in terms of the lower-level type.

In order to complete this implementation program, there is one additional, and quite essential, function which must be provided: the *representation function*. Remember that the operators from each individual type are functions that map into different ranges. It would be absurd to make an implementation program that then equates the range of one ADT's operator with the range of a different ADT's operator. The program only makes sense when there is a representation function which shows how to map an instance of the lower-level type into an instance of the higher-level type. Then, more precisely, we can show how the operators of the higher-level type behave *in terms of the range of the representation function applied to operators of the lower level type.*

Another subtle requirement that is placed on us is to come up with an *equivalence interpretation*. The implementation program is a list of equations, as described above. But remember that, in general, questions of equality are very difficult to resolve. If we are to have any hope of being able to verify properties of this program, then we must have a reliable way to answer questions of equality *in terms of the lower-level type*. The emphasis is for an important distinction: An implementation and its associated equivalence interpretation do not resolve the problem of checking equality within just the higher level type system. They will only allow us to check whether similar operations on instances of the lower-level type will produce (through the representation function) identical sentences in the language of the higher-level type.

We are now ready to summarize what it means to verify an implementation of some ADT. The most obvious requirement is that the axioms for our type of interest are preserved through the implementation. Since the only way we mathematically characterize our ADT is through how they interrelate with one another, the only place we can turn to check the reasonableness of an implementation is exactly this body of rules. These must be checked. A less obvious proof obligation is to verify that our equality interpretation is in fact an equivalence relation. Finally, we must prove a representation invariant, that is, insure that all elements of our ADT domain can in fact be represented as the image of the representation function applied to some instances of the lower level type.

5.3. Example: Verifying an Implementation of Stacks

Using a technique developed by Guttag, Horowitz, and Musser [25], we now illustrate the idea of implementing one ADT in terms of another and then verifying the correctness of that implementation by showing how the ADT stack defined previously may be implemented in terms of arrays. To do this, our first step must be to formulate an axiomatization of array. Its syntax will be:

$$\begin{array}{ll} newarray: & \rightarrow array \\ assign: & array \times integer \times integer \rightarrow array \\ access: & array \times integer \rightarrow integer \cup \{undefined\} \end{array}$$

and its semantics will be simply:

$$\begin{array}{ll} access(newarray, I) = & undefined \\ access(assign(array, I_1, val), I_2) = & if\ I_1 = I_2\ then\ val \\ & \quad else\ (access(array, I_2)) \end{array}$$

Note that this is a somewhat simplistic view of what arrays are. In particular, this is an unbounded array, which is useful only for purposes of introduction. As we discussed in the previous section, it will be important for us to ultimately give our implementation of STACKs in terms of a data type which is in turn more closely implemented directly by the underlying computer.

We choose our representation function to be:

$$STAK(array, integer) \rightarrow STACK$$

and hence we may give our implementation program:

$$
\begin{array}{ll}
newstack = & STAK(newarray, 0) \\
push(STAK(arr, t), x) = & STAK(assign(arr, t+1, x), t+1) \\
pop(STAK(arr, t)) = & if\ t = 0\ then\ STAK(arr, 0) \\
 & \quad else\ (STAK(arr, t-1)) \\
top(STAK(arr, t)) = & access(arr, t) \\
empty(STAK(arr, t)) = & (t = 0)
\end{array}
$$

Our representation invariant can now be declared to be the predicate

$$P(S) \ = \ \exists A \in ARRAY,\ \exists i \in INTEGERS, i \geq 0,\ S = STAK(A, i)$$

where S is a free variable corresponding to an element of STACK.

In order to insure that the correspondence between an element of STACK and a pair $(array, i)$ is reasonable, we must show each of:

$$
\begin{array}{l}
P(newstack) \\
P(S) \Rightarrow P(push(S, x)) \\
P(S) \Rightarrow P(pop(S))
\end{array}
$$

Since we have already proven a normal form lemma for stacks, we only have to prove the first two of these assertions. So:

• Need to show $P(newstack)$, that is:

$$\exists A \in ARRAY, \exists i \in INTEGERS, newstack = STAK(A, i)$$

Based on our implementation, all we need do to show this is simply choose A to be $newarray$, and i to be 0.

• Need to show $P(S) \Rightarrow P(push(S, x))$. We know $S = STAK(A, i) \wedge i \geq 0$ from $P(S)$, and need to show $\exists A'$ and i' such that $push(S, x) = STAK(A', i')$. But $push(S, x) = STAK(assign(A, i+1, x), i+1)$. We can show the desired assertion simply by choosing A' to be $assign(A, i+1, x)$ and i' to be $i+1$, with $i' \geq 0$.

Equality

Our equality interpretation is chosen as follows:

$$STAK(A_1, t_1) = STAK(A_2, t_2)$$

if and only if

$$(t_1 = t_2) \ \wedge \ \forall k(1 \leq k \leq t_1, access(A_1, k) = access(A_2, k))$$

Our proof obligation here is to show that this is a "reasonable" interpretation, that is, it has the properties of being an equivalence relation:

1. $reflexivity$: $x = x$
2. $symmetry$: $x = y \ \Rightarrow \ y = x$
3. $transitivity$: $x = y \ \wedge \ y = z \ \Rightarrow \ x = z$

and, so that we may manipulate expressions in our word algebra safely, we must show that the relation preserves substitutivity, i.e.

$$x = y \;\Rightarrow\; P(x) = P(y)$$

where P can be any of our operators. So, in turn:

• **reflexivity:** We need to show that $S = S$ for a stack S and our equivalence interpretation. We know S is in the range of STAK, based on our representation invariant, that is, $S = STAK(A, t)$. But $A = A$ and $t = t$ based on our usual interpretation of equality (over integers and arrays), hence $STAK(A, t) = STAK(A, t)$ in our desired equality interpretation.

• **symmetry:** We need to show that $S_1 = S_2$ implies $S_2 = S_1$. Since $S_1 = S_2$ under our equivalence interpretation, the arrays and integers which map into these stacks under STAK are themselves equal. Since the equality of integers and arrays is symmetric, then $S_2 = S_1$ is also true in our equality interpretation.

• **transitivity:** Precisely the same argument as given for symmetry will show transitivity holds.

• **substitutivity:** We need to show each of:

$$\begin{aligned}
S_1 = S_2 &\;\Rightarrow\; push(S_1, x) = push(S_2, x) \\
S_1 = S_2 &\;\Rightarrow\; pop(S_1) = pop(S_2) \\
S_1 = S_2 &\;\Rightarrow\; top(S_1) = top(S_2) \\
S_1 = S_2 &\;\Rightarrow\; empty(S_1) = empty(S_2)
\end{aligned}$$

We will show this proof for the push operation, and leave the remaining three as exercises. Note that all operators must be shown in this step — the substitutive property must hold for all operators, even those which map out of the type of interest, hence having proven a normal form lemma does not relieve us of work in this step of the proof.

Let $S_1 = STAK(A_1, i_1)$ and $S_2 = STAK(A_2, i_2)$, and assume that $S_1 = S_2$ to show that

$$push(STAK(A_1, i_1), x) = push(STAK(A_2, i_2), x)$$

Use the implementation of push on the right-hand side of the above expression to obtain

$$STAK(assign(A_1, i_1 + 1, x), i_1 + 1) = STAK(assign(A_2, i_2 + 1, x), i_2 + 1)$$

In terms of our equality interpretation, we must show

$$\begin{aligned}
&(i_1 = i_2) \;\wedge\; \forall k(1 \le k \le i_1, access(A_1, k) = access(A_2, k)) \;\Rightarrow \\
&\quad (i_1 + 1 = i_2 + 1) \;\wedge \forall k(1 \le k \le i_1 + 1, \\
&\quad access(assign(A_1, i_1 + 1, x), k) = access(assign(A_2, i_2 + 1, x), k)
\end{aligned}$$

The first clause of the consequent follows directly from the first clause of the antecedent. The second clause of the consequent is supported by simplifying as follows:

$$access(assign(A_1, i_1 + 1, x), k) = \; if \; k = i_1 + 1) \; then \; x \; else \; access(A_1, k)$$

due to the implementation program. Likewise,

$$access(assign(A_2, i_2 + 1, x), k) = \; if \; k = i_2 + 1) \; then \; x \; else \; access(A_2, k)$$

Hence, because $i_1 = i_2$, the clause we are examining simplifies to

$$\forall k(1 \leq k \leq i_1 + 1,\ if\ k = i_1 + 1\ then\ x = x)$$
$$else\ (access(A_1, k) = access(A_2, k)))$$

For the range $i \leq k \leq i_1$, this is implied by clause two of the antecedent. When $k = i_1 + 1$, we know the expression is true due to the symmetry of "=" over integers.

Finally, we are at the point where we can verify that the axioms of our type STACK still hold through our implementation. The expressions which must be checked are:

1.	***pop(newstack) =***	***newstack***
2.	***pop(push(S, I)) =***	***S***
3.	***top(newstack) =***	***undefined***
4.	***top(push(S, I)) =***	***I***
5.	***empty(newstack) =***	***true***
6.	***empty(push(S, I)) =***	***false***

We will only prove two of these here, leaving the remaining as exercises.

First, we prove Item 4 above, showing

$$top(push(S, x)) = x$$

through the implementation. Using the representation invariant, assume that $S = STAK(A, i)$. Then

$$top(push(STAK(A, i), x)) =$$
$$top(STAK(assign(A, i + 1, x), i + 1)) =$$
$$access(assign(A, i + 1, x), i + 1) = x$$

Thus, the left side reduces to x and we have $x = x$.

Next we prove Item 2 above, showing

$$pop(push(S, x)) = S$$

Again, assume $S = STAK(A, i)$.

$$pop(push(STAK(A, i), x)) =$$
$$pop(STAK(assign(A, i + 1, x), i + 1)) =$$
$$if\ i + 1 = 0\ then\ STAK(assign(A, i + 1, x), 0))$$
$$\quad else\ STAK(assign(A, i + 1, x), i)$$

Now we must use data type induction to show that $P(STAK(A, i))$ implies $i \geq 0$ so that we need consider only one case above.

$$P(newstack) = P(STAK(newarray, 0)) = 0 \geq 0.$$
$$P(S) \Rightarrow P(push(S, x))$$
$$\quad P(STAK(A, i)) \Rightarrow P(push(STAK(A, i), x))$$
$$\quad i \geq 0 \Rightarrow P(STAK(assign(A, i + 1, x), i + 1))$$
$$\quad i \geq 0 \Rightarrow i + 1 \geq 0$$

Because of our normal-form lemma, we do not need to show $P(S) \Rightarrow P(pop(S))$ separately.

Since $i \geq 0$, $i + 1 \neq 0$, so we need only consider

$$STAK(assign(A, i+1, x), i) = STAK(A, i)$$

Using the equality interpretation:

$$(i = i) \wedge \forall k(1 \leq k \leq i, access(assign(A, i+1, x), k) = access(A, k))$$

Using equality of integers and the access implementation, this simplifies to

$$\forall k(1 \leq k \leq i, if\ i+1 = k\ then\ x\ else\ access(A, k) = access(A, k))$$

Since $i + 1 \neq k, \forall k(1 \leq k \leq i, access(A, k) = access(A, k))$, and Item 2 is proved.

5.4. Verifying Applications With ADTs

As stated earlier in this chapter, it is important not only that we be able to verify implementations of abstract data types, but also that we be able to reason about programs which themselves use ADTs. Below is a simple use of our ADT "stack" to effect a "swap" between the variables a and b. Note that we do not need to initialize the variable s to be $newstack$. This program will perform as intended whether or not there happen to be other items already pushed on the stack when the program begins execution.

$$\begin{array}{l}\{true\} \\ \quad s := push(s, a) \\ \quad s := push(s, b) \\ \quad a := top(s) \\ \quad s := pop(s) \\ \quad b := top(s) \\ \{a = \overleftarrow{b} \wedge b = \overleftarrow{a}\}\end{array}$$

Let us now use Hoare-style inference techniques to verify the partial correctness of this program. Our inference system consists of both our existing first order predicate calculus, as enhanced to reason about assignment statements, plus the axioms for dealing with this new type "stack." As usual, we start with the postcondition and work back towards the precondition, using the axiom of assignment:

$$\begin{array}{l}\{a = \overleftarrow{b} \wedge top(s) = \overleftarrow{a}\}\ b := top(s)\ \ \{a = \overleftarrow{b} \wedge b = \overleftarrow{a}\} \\ \{a = \overleftarrow{b} \wedge top(pop(s)) = \overleftarrow{a}\} s := pop(s) \{a = \overleftarrow{b} \wedge top(s) = \overleftarrow{a}\} \\ \{top(s) = \overleftarrow{b} \wedge top(pop(s)) = \overleftarrow{a}\} a := top(s) \{a = \overleftarrow{b} \wedge top(pop(s)) = \overleftarrow{a}\} \\ \{top(push(s, b)) = \overleftarrow{b} \wedge top(pop(push(s, b))) = \overleftarrow{a}\} \\ \quad s := push(s, b) \{top(s) = \overleftarrow{b} \wedge top(pop(s)) = \overleftarrow{a}\} \\ \{top(push(push(s, a), b)) = \overleftarrow{b} \wedge top(pop(push(push(s, a), b))) = \overleftarrow{a}\} \\ \quad s := push(s, a) \\ \{top(push(s, b)) = \overleftarrow{b} \wedge top(pop(push(s, b))) = \overleftarrow{a}\}\end{array}$$

To identify that the above expression which we have derived is in fact equal to the program's precondition, simply apply the axioms for stack.

$$
\begin{array}{l}
\{1 \le n \ \wedge \ \forall j \ni 1 \le j \le n, a[j] = \overleftarrow{a}[j]\} \\
\quad S \ := newstack; i \ := \ 1 \\
\quad \textbf{while } i \le n \textbf{ do} \\
\qquad S \ := push(S, a[i]); i \ := \ i + 1 \\
\quad i \ := \ 1 \\
\quad \textbf{while } i \le n \textbf{ do} \\
\qquad a[i] \ := top(S); S \ := pop(S); i \ := \ i + 1 \\
\{\forall j \ni 1 \le j \le n, a[j] = \overleftarrow{a}[n - j + 1]\}
\end{array}
$$

Figure 5.1. Program to reverse an array, using the ADT "stack."

5.5. Example: Reversing an Array Using a Stack

In Figure 5.1 is our old friend, the "array reversal program" implemented using our abstract data type STACK. We would probably never implement an array reversal in this way, but it is instructive to see the same sort of example in many different contexts. The method chosen is straightforward: All elements are pushed onto the stack in order, then, again in order, are popped off back into the array. A verification that the array is indeed reversed proceeds as follows: The precondition directly implies the program initialization section, as seen in Step 1, which also shows how our choice for the first loop invariant is supported by the initialization. The first loop maintains its invariant, as shown in Step 2. After the first loop, the invariant and $i > n$, followed by execution of $i := 1$, implies the expression which is our choice for the second loop's invariant. This is shown in Step 3. The invariant for the second loop is maintained, as shown in Step 4. Finally, the second invariant and $i > n$, after the second loop, clearly implies our postcondition, and the proof is complete.

In order to describe the remaining details, we accept that the first loop's invariant will be

$$J \ \triangleq \ S = \mathbf{PUSH}(i-1) \ \wedge \ i \le n+1 \ \wedge \ \forall j \ni 1 \le j \le n, a[j] = \overleftarrow{a}[j]$$

and the second loop's invariant will be

$$I \ \triangleq \ \left\{ \begin{array}{l} \forall j \ni 1 \le j < i, a[j] = \overleftarrow{a}[n-j+1] \\ \wedge \, S = \mathbf{PUSH}(n-i+1) \ \wedge i \le n+1 \end{array} \right.$$

where we define the notation

$$\mathbf{PUSH}(j) := \left\{ \begin{array}{ll} (j > 0 \rightarrow & push(\mathbf{PUSH}(j-1), \overleftarrow{a}[j])) \\ \mid (j = 0 \rightarrow & newstack) \end{array} \right.$$

Step One: Initialization

By use of our axiom of assignment we know

$$\underbrace{\{S = \mathbf{PUSH}(0) \wedge 1 \le n+1 \wedge \forall j \ni 1 \le j \le n, a[j] = \overleftarrow{a}[j]\}}_{Z} \ i := 1 \ \{J\}$$

as well as

$$\{\mathbf{true} \wedge \mathbf{precondition}\}\ S := newstack\ \{Z\}$$

which can be composed for our initialization.

Step Two: First loop maintains invariant

By twice applying our axiom of assignment, we know

$$\underbrace{\{S = \mathbf{PUSH}(i)\ \wedge\ i \leq n\ \wedge\ \forall j \ni 1 \leq j \leq n, a[j] = \overleftarrow{a}[j]\}}_{J'}\ i := i + 1\ \{J\}$$

and

$$\underbrace{\{push(S, a[i]) = \mathbf{PUSH}(i)\ \wedge\ i \leq n\ \wedge\ \forall j \ni 1 \leq j \leq n, a[j] = \overleftarrow{a}[j]\ \}}_{J''}$$
$$S := push(S, a[i])\ \{J'\}$$

After composing these, we observe that $J \wedge (i \leq n)$ implies clause two of J'' directly; it implies clause one by simple use of our notation; and it implies clause three directly. Hence, the invariant is maintained.

Step Three: Output of first loop implies input to second loop

The result of the first loop is

$$\begin{aligned} S = \mathbf{PUSH}(i-1) \wedge (i \leq n+1) \wedge \neg(i \leq n) \\ \Rightarrow S = \mathbf{PUSH}(n) \\ \Rightarrow S = \mathbf{PUSH}(n) \wedge \mathbf{true} \\ \Rightarrow S = \mathbf{PUSH}(n) \wedge 1 = 1 \end{aligned}$$

which can be used in a rule of consequence with the following application of the axiom of assignment

$$\{S = \mathbf{PUSH}(n) \wedge 1 = 1\}\ i := 1\ \{S = \mathbf{PUSH}(n) \wedge i = 1\}$$

This expression implies the first clause of I vacuously; it implies the second clause by simple rearrangement of notation and repeated use of our algebraic rules for the ADT; and it implies the third clause of I directly, assuming we were clever enough to have pulled the requirement $n \geq 1$ down through the program so we could use it at this point. The third clause of J allows us to substitute $\overleftarrow{a}$ for our uses of a in the expression involving S. Hence, we have established I at the start of the second loop.

Step Four: Second loop maintains Invariant

Repeatedly using our axiom of assignment,

$$\underbrace{\begin{array}{c}\{\forall j, 1 \le j < i+1, a[j] = \overleftarrow{a}[n-j+1] \\ \wedge\ S = \mathbf{PUSH}(n-(i+1)+1)\ \wedge\ i+1 \le n+1\}\end{array}}_{I'} \quad i := i+1\ \{I\}$$

$$\underbrace{\begin{array}{c}\{\forall j \ni 1 \le j < i+1, a[j] = \overleftarrow{a}[n-j+1] \\ \wedge\ pop(S) = \mathbf{PUSH}(n-(i+1)+1)\ \wedge\ i \le n\}\end{array}}_{I''} \quad S := pop(S)\ \{I'\}$$

$$\underbrace{\begin{array}{c}\{\forall j \ni 1 \le j < i+1, \alpha(a,i,top(S))[j] = \overleftarrow{a}[n-j+1] \\ \wedge\ pop(S) = \mathbf{PUSH}(n-(i+1)+1)\ \wedge\ i \le n\}\end{array}}_{I'''}$$

$$a[i] := top(S)\{I''\}$$

then composing these expressions we obtain:

$$\{I'''\} \quad body\ of\ while \quad \{I\}$$

We claim that $I \wedge (i \le n) \Rightarrow I'''$ hence showing that the second **while** loop invariant is maintained: For j in the range $1 \le j < i$, clause one of I''' follows directly from the first clause of I. For $j = i$, I''' reduces as follows:

$$\begin{array}{l} I \Rightarrow\ top(S)\ =\ top(\mathbf{PUSH}(n-i+1)) \\ = top(push(\mathbf{PUSH}(n-i+1-1), \overleftarrow{a}[n-i+1]))\ =\ \overleftarrow{a}[n-i+1] \end{array}$$

The second clause of I''' is supported as follows:

$$\begin{array}{rl} I \Rightarrow & S = \mathbf{PUSH}(n-i+1)\ \Rightarrow\ pop(S) \\ & =\ pop(\mathbf{PUSH}(n-i+1)) \\ & =\ pop(push(\mathbf{PUSH}(n-i+1-1), \overleftarrow{a}[n-i+1])) \\ & =\ \mathbf{PUSH}(n-i+1-1) \\ & =\ \mathbf{PUSH}(n-(i+1)+1) \end{array}$$

The third clause of I''' is directly supported by the loop conditional $i \le n$.

6. INDUCTIONLESS INDUCTION

Previously we have used data type induction to prove assertions concerning our algebraically specified ADTs. There is a second technique which is useful, called *inductionless induction* (also called *structural induction*). It is useful in general, but we will use it here as an essential technique in showing that sets of axioms are consistent (as discussed earlier in this chapter).

The technique is based upon the technique known as unification and on the Knuth-Bendix algorithm for showing consistency among a set of axioms. We first describe unification and then give this algorithm.

6.1. Knuth–Bendix Algorithm

The idea behind this algorithm is fairly simple. The use of rewrite rules separates the set of valid expressions into discrete equivalence classes. If the rules are noetherian and confluent, then applying

rewrite rules for any equation causes the process to terminate with a unique normal-form value. Thus for two expressions, we can determine whether they are in the same class by computing the normal-form value for each.

Using a process called *unification*, described below, we can combine two rewrite rules into a new rule showing the equivalence of two new expressions. If these come from two distinct equivalence classes (computed by forming each's normal-form value), then we have shown that these two classes really are the same and we can collapse them together. We call these two expressions *critical pairs* and can add them as a new rewrite rule in our system.

As long as we add critical pairs, we collapse discrete sets together. However, if we end up by collapsing the equivalence classes described by *true* and *false*, then we have shown the inconsistency of our rewrite rules.

Unification

Unification is an algorithm that forms the basis of Logic programming languages, theorem provers, and type checking systems in functional languages. To unify two expressions U, V means to find substitutions for the variables occurring in U and V so that the two expressions become identical.

For example, unifying $f(X, Y)$ with $f(g(Y), Z)$ is done by binding: X to $g(Y)$, and Y to Z.

Unification can be seen as an algorithm for solving a system of equations. In setting up the algorithm to unify expressions $U = V$, a list, L, of unifications yet-to-be-done is set up with $(U = V)$ as its only member. Each variable, X, occurring in U and V, is set to point to an equation, $EQ(X)$, of the form:

$$Variable - List = Expression : Occurrences$$

where $Occurrences$ keeps track of the number of times any variable on the left-hand side occurs on the right-hand side of any other equation. This will keep track of cyclic references.

The Knuth–Bendix algorithm is basically a search technique, where each axiom (along with cleverly chosen transformations of that axiom) is compared to all the remaining axioms in order to check whether an inconsistency has arisen. There are three possible outcomes:

1. The algorithm terminates (possibly after generating new rules none of which are inconsistent). In this case, the set of axioms is consistent. In Section 6.2 we use this fact to give another proof technique (inductionless induction) along with the already given data type induction.

2. An inconsistency is discovered. This usually presents itself by deriving the rule $true = false$ from the existing set of rules.

3. The algorithm does not terminate and an infinite set of rules is generated. In this case, the Knuth-Bendix algorithm cannot determine the consistency of the set of rules.

Huet's version [32] of the Knuth–Bendix algorithm is shown below. In order to use the algorithm, we must be able to orient the equations of the form $x = y$ into rewrite rules $x \rightarrow y$. Huet and Oppen describe a method for weighing words that permits equations to be ordered so that rewrite rules reduce the weights of equations.

Weighing words

The weight of a word w is defined as

$$weight(w) \;=\; weight_0 \;\times\; \sum_{j=1}^{V} occurs(v_j, w) \;+\; \sum_{j=1}^{O} weight_j \times occurs(f_j, w)$$

where $weight_0$ is the minimum weight of a nullary operator, v_j is a variable, $occurs(n, w)$ is the number of occurrences of symbol n in word w, and $weight_j$ is the weight of operator f_j. The symbols O and V represent the number of operators and number of variables respectively.

For example, consider natural numbers defined by the axioms:

$$\begin{array}{ll} add(0, X) = & X \\ add(succ(X), Y) = & succ(add(X, Y)) \end{array}$$

and the equation describing the associativity of addition

add(add(X,Y),Z) = add(X,add(Y,Z))

Generally in defining operators' weights, f is assigned a higher weight than g if f is used to define g. Also unary operators are usually assigned $weight_0$. The arithmetic operators have the following weights:

add	0
succ	0
0	1

Since 0 is the only nullary operator, $weight_0$ is 1. However, ***succ*** is not assigned a similar weight. Some sample words and their weights are shown below:

Word	Weight	
X	$1 \times (1) + (\,(0 \times 0) + (0 \times 0) + (1 \times 0)\,) =$	1
add(0,X)	$1 \times (1) + (\,(0 \times 1) + (0 \times 0) + (1 \times 1)\,) =$	2
add(***succ***(X),Y)	$1 \times (2) + (\,(0 \times 1) + (0 \times 1) + (1 \times 0)\,) =$	2
add(***succ***(X,Y))	$1 \times (2) + (\,(0 \times 1) + (0 \times 1) + (1 \times 0)\,) =$	2
add(***add***(X,Y),Z)	$1 \times (1+1+1) + (\,(0 \times 2) + (0 \times 0) + (1 \times 0)\,) =$	3
add(X,***add***(Y,Z))	$1 \times (1+1+1) + (\,(0 \times 2) + (0 \times 0) + (1 \times 0)\,) =$	3

Ordering equations

The relation used for ordering words $a > b$ is defined as follows: $a > b$ iff either

1. $weight(a) > weight(b)$ and $1 \le i \le V \;\Rightarrow\; occurs(v_i, a) \ge occurs(v_i, b)$; or

2. $weight(a) = weight(b)$ and $1 \le i \le V \;\Rightarrow\; occurs(v_i, a) = occurs(v_i, b)$, and either

(a) $a = f(v_k)$ and $b = v_k$ for some operator f, or

(b) $a = f_j(a_1, \ldots, a_j)$ and $b = f_k(b_1, \ldots, b_k)$ and either $j > k$, or both $j = k$ and $1 \leq t \leq j \Rightarrow a_1 = b_1, \ldots, a_{t-1} = b_{t-1}, a_t > b_t$.

Several examples appear below:

Words	Reasons
add(0,X) > X	2>1
add(***succ***(X),Y) > ***succ***(***add***(X,Y))	2=2, each has 2 variables, first has fewer args
add(***add***(X,Y),Z) > ***add***(X,***add***(Y,Z))	3=3, each has 3 variables, ***add***(a,b) > a
add(***succ***(X),X) ? ***add***(0,X)	2=2, but the number of variables differs

In the algorithm, the following operations use the notion of order:

Operation	Meaning
orderable$(s = t)$	$(s > t)$ or $(t > s)$
order$(s = t)$	$(s > t \rightarrow (s = t)) \mid (t = s)$

Here, ***orderable*** is a Boolean-valued predicate that checks whether the two words *can* be compared according to the above criteria. ***Order*** is a function which then would return the rewrite rule based on the original equation.

Critical pairs

Critical pairs arise when (with renaming of variables) the left side of one rewrite rule occurs as an argument in the left side of another rule. This is an application of the unification property described earlier. Thus equations that could be reduced by the second rule could also be reduced by the first. After applying unification, if the normal forms for each expression in the rewrite rule are different, then we have a critical pair. The critical pair produced is an equation asserting the equality of the two expressions.

Some notation must first be developed in order to discuss critical pairs further:

Notation	Meaning
t/u	The subterm u of t
t^u_y	Replace subterm u of t with y
σ	A substitution $\{u := y\}$ applied to a term
$normalize(x)$	Compute normal form for expression x

Superposition algorithm

Assume rewrite rules $u \rightarrow v$ and $x \rightarrow y$ with the property that u and x have no variables with common names. If z is a nonvariable occurrence in u such that u/z and x are unifiable with minimal unifier σ, then the pair of terms $\langle \sigma(u^z_y), \sigma(v) \rangle$ is potentially a critical pair. If $normalize(\sigma(u^z_y)) \neq normalize(\sigma(v))$, then they are a critical pair, and they represent an additional rule (or theorem) derivable from our axioms: $normalize(\sigma(u^z_y)) = normalize(\sigma(v))$. (Actually, we need the terms in the correct order, or $order(normalize(\sigma(u^z_y))) = order(normalize(\sigma(v)))$.)

For example, the rules:

$$add(\mathbf{0}, W) \rightarrow W$$
$$add(add(X, Y), Z) \rightarrow add(X, add(Y, Z))$$

with unifier:

$$\sigma(X := \mathbf{0}, Y := W)$$

produce the critical pair:

$$\langle add(W, Z), add(\mathbf{0}, add(W, Z)) \rangle$$

Using this notation, the Knuth-Bendix algorithm is given in Figure 5.2. An example of the use of this algorithm is given in the next section.

6.2. Application of Knuth Bendix to induction

The idea of inductionless induction was introduced by Musser [46] [47] when he proved the following theorem:

Theorem: Let T be a collection of types with axiom set A and assume that T is fully specified by (i.e., sufficiently complete with respect to) A. If E is a set of equations, each in the language generated by T, and A ∪ E is consistent, then each equation in E is in the inductive theory of T.

Thus to prove an equation $x = y$ by inductionless induction, add it to a set of equations containing a sufficiently complete set of axioms and execute the Knuth–Bendix algorithm. If the algorithm terminates, then $x = y$ is consistent with the previous axioms; if it derives $true = false$, then the new rule is inconsistent with the set of axioms.

As an example (adapted from [17]) of both inductionless induction and the use of Knuth–Bendix, we now use this method to show how the axiom of associativity of addition is consistent with the algebraic axioms of addition given earlier. Initially, the set of equations E contains the following members:

$$add(\mathbf{0}, U) = U$$
$$add(succ(V), W) = succ(add(V, W))$$
$$add(add(X, Y), Z) = add(X, add(Y, Z))$$

The **while** statement guarded by $E \neq \{\}$ removes these equations from E one at a time, orients them according to the weighing scheme described above, normalizes them with respect to each other, and adds them to R (the list of rewrite rules):

$$\langle add(\mathbf{0}, U) \rightarrow U, \mathbf{1}, unmarked \rangle$$
$$\langle add(succ(V), W) \rightarrow succ(add(V, W)), \mathbf{2}, unmarked \rangle$$
$$\langle add(add(X, Y), Z) \rightarrow add(X, add(Y, Z)), \mathbf{3}, unmarked \rangle$$

After E is empty, the **for** statement selects unmarked rules from R, computes critical pairs for lower numbered members of R, adds the critical pairs to E, and replaces the unmarked rules with marked rules. For our example, no critical pairs are computed until the third element of R is selected.

```
R := {};
i := 0;
done := false;
while ¬done do begin
    done := true;
    while E ≠ {} do begin      "Order E's equations and put them in R."
        "Find non – joinable critical pair"
        (s = t) := choseOneFrom(E);
        E := E – {s = t};
        s' := normalize(s, R); t' := normalize(t, R)
        if s' ≠ t' then begin "Order equation."
            if ¬orderable(s' = t') then return(fail);
            (v → w) := order(s' = t');
            "Normalize rewriting system."
            for each ⟨x → y, positionXY, markXY⟩in R do begin
                x' := normalize(x, {v → w});
                if x ≠ x' then begin
                    R := R – {⟨x → y, positionXY, markXY⟩};
                    E := E ∪ {x' = y}
                    end
                else begin
                    y' := normalize(y, R ∪ {v → w});
                    if y ≠ y'then
                        R := (R – {⟨x → y, positionXY, markXY⟩})
                        ∪{⟨x → y', positionXY, markXY⟩}
                    end
                end;"... for each"
            i := i + 1;
            R := R ∪ {⟨v → w, i, unmarked⟩}
            end"... order equation."
    end;"... while E ≠ {}"
    "Find an unmarked rule."
    for each⟨x → y, positionXY, markXY⟩in R do
        if markXY = unmarked then begin
            "Compute critical pairs."
            done := false;
            for each⟨u → v, positionUV, markUV⟩in R do
                if positionUV ≤ positionXY then
                    E := E ∪ criticalpairs(x → y, u → v);
            R := (R – {⟨x → y, positionXY, markXY⟩})∪
                {⟨x → y, positionXY, marked⟩}
            end;
    end;"... while not done."
return(success).
```

Figure 5.2. Knuth–Bendix Algorithm

$$\langle add(\mathbf{0}, U) \rightarrow U, \mathbf{1}, marked\rangle$$
$$\langle add(succ(V), W) \rightarrow succ(add(V, W)), \mathbf{2}, marked\rangle$$
$$\langle add(add(X, Y), Z) \rightarrow add(X, add(Y, Z)), \mathbf{3}, unmarked\rangle$$

The following critical pairs are computed and added to E:

Unifier	**Equation**
$\{X := \mathbf{0}, Y := U\}$	$add(U, Z) = add(\mathbf{0}, add(U, Z))$
$\{X := succ(V), Y := W\}$	$add(succ(add(V, W)), Z) =$ $add(succ(V), add(W, Z))$

The final element of R is also marked so no unmarked members of R remain. Since *done* has value *false* the outer *while* statement is executed again with E no longer empty. When one of the equations of E is selected and normalized with respect to R, E is empty and no new rewrite rules are added to R. Thus the algorithm terminates.

$add(U, Z) = add(\mathbf{0}, add(U, Z))$:

$$normalize(add(U, Z)) = add(U, Z)$$
$$normalize(add(\mathbf{0}, add(U, Z))) = add(U, Z) \ \ (by\ 1)$$

$add(succ(add(V, W)), Z) = add(succ(V), add(W, Z))$:

$$normalize(add(succ(add(V, W)), Z)) =$$
$$\quad succ(add(add(V, W), Z)) = \ \ (by\ 2)$$
$$\quad succ(add(V, add(W, Z))) = \ \ (by\ 3)$$
$$normalize(add(succ(V), add(W, Z)) =$$
$$\quad succ(add(V, add(W, Z))) \ \ (by\ 2)$$

Notice that equations like the commutativity of addition "***add***(X,Y) = ***add***(Y,X)" cannot be proved using this technique because the terms cannot be ordered by weight. However, such equations can be proved by data type induction. (This is given as an exercise.)

6.3. Example Using Knuth–Bendix

Consider the following specification for sequences of integers. The symbols S, S', T, and T' will represent sequences, and x and x' will represent integers. We will define the specifications for five functions:

1. $null$ returns the null sequence.
2. $S + x$ appends integer x onto the end of sequence S.
3. $x|S$ places x at the start of sequence S.
4. $S\ cat\ T$ concatenates two sequences S and T into one sequence.
5. $isnull(S)$ returns $true$ if sequence S is null; $false$ otherwise.

The signatures for these functions are:

$$\begin{array}{ll} null & \rightarrow sequence \\ + & sequence \times integer \rightarrow sequence \\ | & integer \times sequence \rightarrow sequence \\ cat & sequence \times sequence \rightarrow sequence \\ isnull & sequence \rightarrow boolean \end{array}$$

The ten axioms needed to define these sequences are:

$$\begin{array}{ll} Ax_1 & (S = S) = true \\ Ax_2 & (null = S + x) = false \\ Ax_3 & (S + x = null) = false \\ Ax_4 & (S + x = S' + x') = ((S = S') \wedge (x = x')) \\ Ax_5 & x|null = null + x \\ Ax_6 & x|(S + x') = (x|S) + x' \\ Ax_7 & null\ cat\ S = S \\ Ax_8 & (S + x')\ cat\ S' = S\ cat\ (x'|S') \\ Ax_9 & isnull(null) = true \\ Ax_{10} & isnull(S + x) = false \end{array}$$

Example: Data type induction

Let us review the previous method of data type induction by proving that

$$S\ cat\ (S' + x') = (S\ cat\ S') + x'$$

.

Let $P(S)$ be the predicate: $\forall S'(S\ cat\ (S' + x') = (S\ cat\ S') + x')$. We will assume a normal form lemma, such as with stacks, that states that the constructors for $sequences$ are $null$ and $+$. Both cat and $|$ can be obtained by applications of the first two functions. To use data type induction, we need to show that:

1. $P(S)$, and
2. $P(S) \Rightarrow P(S + x)$

(1) $P(null) = \forall S'(null\ cat\ (S' + x') = (null\ cat\ S') + x')$

Proof:

$$\begin{array}{ll} null\ cat\ (S' + x') = (null\ cat\ S') + x' & Hypothesis \\ (S' + x') = (null\ cat\ S') + x' & Ax_7 \\ (S' + x') = (S' + x') & Ax_7 \\ true & Ax_1 \end{array}$$

□

(2) Show $P(S) \Rightarrow P(S + x)$.

Proof:

$$
\begin{array}{lll}
Assume\ P(S)\ =\ S\ cat\ (S' + x')\ = & & \\
\quad (S\ cat\ S') + x' & & \\
Show\ P(S + x)\ =\ (S + x)\ cat\ (S' + x')\ = & & \\
\quad ((S + x)\ cat\ S') + x' & & \\
(S + x)\ cat\ (S' + x')\ =\ ((S + x)\ cat\ S') + x' & & \\
S\ cat\ (x|(S' + x'))\ =\ ((S + x)\ cat\ S') + x' & Ax_8 & \\
S\ cat\ ((x|S') + x')\ =\ ((S + x)\ cat\ S') + x' & Ax_6 & \\
(S\ cat\ (x|S')) + x'\ =\ ((S + x)\ cat\ S') + x' & Inductive\ Hypothesis & \\
(S\ cat\ (x|S')) + x'\ =\ (S\ cat\ (x|S')) + x' & Ax_8 & \\
true & Ax_1 & \\
\square & &
\end{array}
$$

Thus we have shown $P(S)$ for all $S \in sequences$.

Using Knuth–Bendix

There are four conditions that can result from applying the Knuth–Bendix algorithm to a set of axioms. In one case, given axiom $\alpha = \beta$, we are unable to define a weight function w such that $w(\alpha) > w(\beta)$. In this case, the algorithm does not apply and we cannot use it to determine the consistency of the axioms.

In the case where we can order the axioms, the algorithm has three results:

1. The algorithm cannot develop any more critical pairs. In this case, we have shown that the axioms are consistent. If an original set of axioms is complete and we add a new axiom, then this new axiom must be consistent with the complete set of axioms, and hence must be a theorem provable from the original axioms.

2. The algorithm derives the critical pair $true = false$. This is a termination condition which shows that the original axioms are inconsistent.

3. The algorithm generates a nonterminating set of critical pairs. In this case, the algorithm is inconclusive as to the correctness of the set of axioms.

The following examples all demonstrate these results.

Assume the definition of $sequences$ given above with its ten axioms. Add to these axioms the following:

1. $S\ cat\ (S' + x')\ =\ (S\ cat\ S') + x'$. This rule causes the Knuth–Bendix algorithm to terminate. As was already shown above, this statement is consistent with the given axioms.

2. $S\ cat\ null\ =\ S$ and Rule 1 above. These two are consistent and the algorithm terminates.

3. $S\ cat\ null\ =\ S$ without Rule 1. In this case, as will be shown below, the algorithm does not terminate. Without the above "associative" rule, this second rule cannot be proven within the framework of the given ten axioms, even though we know it is true by first proving the companion associative rule.

4. $S\ cat\ null\ =\ null$. Intuitively, this should be $false$. As will be shown, the algorithm demonstrates that this rule is inconsistent with the axioms.

Nontermination example

Consider the ten axioms and the new rule $S\ cat\ null\ =\ S$ (from [47]). Applying Knuth–Bendix gives the following steps:

$T\ cat\ null\ =\ T$	$Substitute\ for\ new\ rule$
$(S + x')\ cat\ S'\ =\ S\ cat\ (x'\|S')$	Ax_8
$\langle S\ cat\ (x'\|null), S + x'\rangle$	$Possible\ Critical\ Pair$
	$\sigma(T := S + x', S' := null)$
$\langle S\ cat\ (null + x), S + x'\rangle$	$Normalize\ expressions\ with\ Ax_5$
$S\ cat\ (null + x)\ =\ S + x$	$Critical\ Pair\ -$
	$New\ rule\ added$

At this point, we can repeat the process. Take the final axiom (critical pair 2, above) and repeat the same five steps using for σ in critical pair 1 the sequence: $S := null, S := (null + x), S := ((null + x) + x', \ldots$. This gives the infinite sequence of critical pairs:

$$
\begin{aligned}
&S\ cat\ (null + x')\ =\ S + x'\\
&S\ cat\ ((null + x) + x')\ =\ (S + x) + x'\\
&S\ cat\ (((null + x) + x') + x''\ =\ ((S + x) + x') + x''\\
&\vdots
\end{aligned}
$$

and the algorithm does not terminate.

Inconsistent axiom example

Consider the rule $S\ cat\ null\ =\ null$. We will show it is inconsistent as follows:

$T\ cat\ null\ =\ null$	$Substitute\ for\ new\ rule$
$(S + x')\ cat\ S'\ =\ S\ cat\ (x'\|S')$	Ax_8
$\langle S\ cat\ (x'\|null), null\rangle$	$Possible\ CriticalPair-$
	$\sigma(T := S + x', S' := null)$
$S\ cat\ (null + x)\ =\ null$	$Critical\ Pair\ 1-$
	$Normalize\ with\ Ax_5$
$null\ cat\ T\ =\ T$	$Substitute\ for\ Ax_7$
$\langle null + x, null\rangle$	$Possible\ Critical\ Pair-$
	$\sigma(T := null + x, S := null)$
$null + x\ =\ null$	$Critical\ Pair\ 2$
$isnull(S + x)\ =\ false$	Ax_{10}
$isnull(null)\ =\ false$	$\sigma(S := null)$
$isnull(null)\ =\ true$	Ax_9
$true\ =\ false$	$Critical\ Pair$

Since we have added the rule $true\ =\ false$, the initial rule is inconsistent with our axioms.

7. EXERCISES

Problem 1. Redo the algebraic specification of a stack in Section 3 so that the definition is modified to replace the top element of the stack with the new element when the stack is full, rather than ignore the new element, as is presently given.

Problem 2. Consider the set of natural numbers with operations: 0, succ, and add.

```
add(X,0)        =  X
add(X,succ(Y))  =  succ(add(X,Y))
```

- Give an appropriate, complete and non-redundant definition of the predicate even(X).

- Prove the following theorem: even(X) $\wedge$ even(Y) $\Rightarrow$ even(add(X,Y))

Problem 3. Consider the type "set of integers" defined by the following axioms.

```
in(emptySet,X)            =  false
in(insert(S,I),J)         =  I=J ∨ in(S,J)
delete(emptySet,X)        =  emptySet
delete(insert(S,I),J)     =  if I=J then delete(S,J)
                                 else insert(delete(S,J),I)
isEmptySet(emptySet)      =  true
isEmptySet(insert(S,X))   =  false
```

- Give a correct, complete, and non-redundant list of axioms for set equality.

- Prove the theorem in(delete(S,X),X) = false.

Problem 4. Consider a data type list with operations:

```
append(null,X)        =  X
append(cons(X,Y),Z)   =  cons(X,append(Y,Z))
```

- Add axioms for the operation rev, which reverses the order of the elements in the list.

- Show rev(rev(X)) = X.

Problem 5. Consider the following definition of type list in which the operation add appends values to the end of a list.

front(newlist)	=	0
front(add(newlist,A))	=	A
front(add(add(L,A),B))	=	front(add(L,A))
tail(newlist)	=	newlist
tail(add(newlist,A))	=	newlist
tail(add(add(L,A),B))	=	add(tail(add(L,A)),B)
length(newlist)	=	0
length(add(L,A))	=	succ(length(L))

• Give axioms to define the operations sorted and perm having the following intuitive definitions. You may define hidden functions if necessary.

- sorted: list $\rightarrow$ boolean. sorted returns true if the list elements appear in ascending order with the smallest value at the front. Otherwise sorted returns false.
- perm: list $\times$ list $\rightarrow$ boolean. perm returns true if one of its arguments is a permutation of the other. Otherwise perm returns false.

Problem 6. Consider the operations member and sinsert, defined as follows:

member(newlist,X)	=	false
member(add(L,X),Y)	=	((X=Y) or member(L,Y))
sinsert(newlist,A)	=	add(newlist,A)
sinsert(add(L,A),B)	=	if B >= A then add(add(L,A),B) else add(sinsert(L,B),A))

Prove the theorem: member(sinsert(L,X),X) = true. You may assume the existence of a normal form lemma showing that all list objects can be constructed from the operations newlist and add.

Problem 7. Give algebraic specifications for a library of books (whose titles are natural numbers) with the following operations:

init:	$\rightarrow$ library	open for business
donate:	library $\times$ nat $\rightarrow$ library	add a volume to library
remove:	library $\times$ nat $\rightarrow$ library	destroy a book & forget it was donated
check:	library $\times$ nat $\rightarrow$ library	borrow a book
return:	library $\times$ nat $\rightarrow$ library	bring a book back
given:	library $\times$ nat $\rightarrow$ bool	has a book been donated?
avail:	library $\times$ nat $\rightarrow$ bool	is book available for checkout?
out:	library $\times$ nat $\rightarrow$ bool	is book already checked out?

Using data type induction, show: given(L,B) $\supset$ (avail(L,B) $\vee$ out(L,B)).

Problem 8. The low-water-mark problem is defined for a system composed of processes and objects, each with its own security level. Three security levels are linearly ordered: classified $\leq$ secret $\leq$ topsecret. Processes have fixed security levels, but objects' security levels may be changed by operations. When a process writes a value in an object, the security level of the object drops to that of the process.

When a process resets an object, the security level of the object becomes top secret and its value becomes undefined. The low-water-mark idea is that the security level of an object may decrease, but not increase unless a reset operation is executed for the object.

The Bell and LaPadula simple security and * properties (colloquially known as "no read up, no write down") require enforcement of the following restrictions:

- For a process to read an object's value, the security level of the process must be greater than or equal to that of the object.
- For a process to write an object's value, the security level of the process must be less than or equal to that of the object.

Using the following operations, give an algebraic specification for the data type state which enforces the Bell and LaPadula properties. Operations attempting to "read up" should return the undefined value. Operations attempting to "write down" should be ignored. Reset operations enforce the same security-level restrictions as write operations.

newstate:	$\rightarrow$ state
reset:	process $\times$ object $\times$ state $\rightarrow$ state
write:	process $\times$ object $\times$ nat $\times$ state $\rightarrow$ state
read:	process $\times$ object $\times$ state $\rightarrow$ nat $\cup$ {undefi ned}
name:	object $\rightarrow$ id
level:	object $\times$ state $\rightarrow$ (classified, secret, topsecret)
level:	process $\rightarrow$ (classified, secret, topsecret)

Problem 9. Consider the list data type with operations:

$$\begin{array}{ll} newlist: & \rightarrow list \\ addelt: & list \times nat \rightarrow list \\ tail: & list \rightarrow list \\ head: & list \rightarrow nat \\ last: & list \rightarrow nat \\ length: & list \rightarrow nat \end{array}$$

with their usual meanings. Give three different sets of axioms defining lists in which elements are added to the right end of the list, in which elements are added to the left end of the list, and in which elements are added so that the list remains sorted in ascending order. To make things a bit easier, you may define operations so that applying them to empty lists yields 0.

Problem 10. If we added the operation join to lists in which elements are added to the right end of the list, we might write the axioms:

$$\begin{array}{ll} Signature: & \\ join: & list \times list \rightarrow list \\ & \\ Axioms: & \\ join_1: & join(L, newlist) = L \\ join_2: & join(L, addelt(M, X)) = addelt(join(L, M), X) \end{array}$$

Show that $join(newlist, L) = L$ is not an axiom, but a theorem following from the given axioms.

Problem 11. Consider the following implementation of the data type SetType with operations NewSet, AddElt, and Member. Give a representation function for the type and write comments for each operation giving pre, post, in and out.

```
const
   SetMax = 256;
type
   EltType =  integer;
   SetIndex = 1..SetMax;
   SetType =  record
                 Next: 0..SetMax;
                 V: array [SetIndex] of EltType
              end;

procedure NewSet(var S: SetType);
begin
   S.Next := 0
end;

procedure AddElt(var S: SetType; E: EltType);
var i: integer;
begin
   S.V[S.Next+1] := E;
   i := 1;
   while (i<=S.Next+1) and (S.V[i]<>E) do i := i+1;
   if i = S.Next+1 then S.Next := S.Next+1
end;

function Member(var S: SetType; E: EltType): boolean;
var I: SetIndex;
    B: boolean;
begin
   B := false;
   for I := 1 to S.Next do B := B or (E = S.V[I]);
   Member := B
end;
```

Problem 12. A bag or multiset is a set that permits duplicate values. Consider the following implementation of the data type "bag of integers." A set cannot be used to represent a bag, as there is no way to indicate the presence in a set of more than one occurrence of an element. Two good choices would be to represent a bag as a sequence of elements, since a sequence can contain duplicated elements, or a set of pairs representing (element, number of occurrences). Which choice makes the specification task easier?

```
const Max = ...;
type T = integer;
     Bag = array [0..Max] of T;
```

```
procedure BagInit(var B: Bag);
   var i: integer;
   begin
      for i := 0 to Max do B[i] := 0
   end;
procedure BagAdd(var B: Bag; X: T);
   begin
      if (0 <= X) and (X <= Max) then B[X] := B[X] + 1
   end;
procedure BagDel(var B: Bag; X: T);
   begin
      if (0 <= X) and (X <= Max) and (B[X] > 0)
         then B[X] := B[X] - 1
   end;
function BagMember(var B: Bag; X: T): boolean;
   begin
      if (0<=X) and (X<=Max) then BagMember := B[X]>0
      else BagMember := false
   end;
```

Problem 13. Give a representation function for the type and abstract and concrete comments for data type IntSet.

```
type IntSet = array [1..Max] of boolean;

procedure IntSetInit(var S: IntSet);
var i: integer;
begin
   for i := 1 to Max do S[i] := false
end;

procedure IntSetAdd(var S: IntSet; X: integer);
begin
   if (1<=X) and (X<=Max) then S[X] := true
end
```

Problem 14. The following code implements type "queue of EltType" as a circular list with a contiguous representation.

```
const Max = 2;
type
   EltType = 0..maxint;
   QueueIndex = 0..Max;
   Queue = record
             M: array [QueueIndex] of EltType;
             H, T: QueueIndex
           end;
```

```
procedure NewQ(var Q: Queue);
begin
   Q.H := 0; Q.T := 0;
end;

procedure EnQ(var Q: Queue; Elt: EltType);
var Temp: QueueIndex;
begin
   Temp := (Q.T + 1) mod (Max + 1);
   if Temp <> Q.H then begin
      Q.T := Temp;
      Q.M[Q.T] := Elt
      end;
end;

procedure DeQ(var Q: Queue; var Result: EltType);
begin
   if Q.H <> Q.T then begin
      Q.H := (Q.H + 1) mod (Max + 1);
      Result := Q.M[Q.H];
      end
   else Result := 0
end;
```

State a representation function for the type. State non-trivial concrete and abstract invariants for the type and show that the concrete invariant implies the abstract invariant. Give pre, post, in, and out comments for NewQ and demonstrate their consistency.

8. SUGGESTED READINGS

The algebraic specification techniques are well-described in the papers:

• J. V. Guttag and J. J. Horning, "The algebraic specification of abstract data types," *Acta Informatica*, Vol. 10, 1978, pp. 27-52.

• J. V. Guttag, "Notes on Type Abstraction (Version 2)," *IEEE Transactions on Software Engineering*, Vol. 6, No. 1, 1980, pp. 13-23.

• J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract Data Types and Software Validation ," *Communications of the ACM*, Vol. 21, No. 12, 1978, pp. 1048-1064.

• J. V. Guttag, E. Horowitz, and D.R. Musser, "The Design of Data Type Specifications ," R.T. Yeh (Ed.), *Current Trends in Programming Methodology 4: Data Structuring*, Prentice-Hall, Englewood Cliffs NJ, 1978, pp. 60-79.

A paper which summarizes the verification process for algebraically specified data types is the following. This paper presents a straightforward technique for expressing algebraic axioms as a pair of pre- and postconditions suitable for Hoare-style proofs.

• L. Flon and J. Misra, "A Unified Approach to the Specification and Verification of Abstract Data Types," *Proceedings of the IEEE Conference on Specifications of Reliable Software*, Cambridge, MA, 1979, pp. 162-169.

Papers which describe proof techniques for algebraic data types include the following:

• D. R. Musser, "Abstract data type specifications in the AFFIRM system," *IEEE Specifications of Reliable Software Conference*, Cambridge MA, 1979, pp. 47-57.

• D. R. Musser, "On Proving Inductive Properties of Abstract Data Types," *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, 1980, pp. 154-162.

• G. Huet, "Confluent Reductions: Abstract Properties and Applications to Term-Rewriting Systems," *Journal of the ACM*, Vol. 27, 1980, pp. 797-821.

• G. Huet and J.-M. Hullot, "Proofs by Induction in Equational Theories with Constructors ," *Journal of Computer and System Science*, Vol. 25, 1982, pp. 239-266.

The first two Musser papers describe the application of the Knuth–Bendix algorithm to the problem of proving properties of ADTs – in this case by describing the AFFIRM system, which Musser built.