IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 4, NO. 20, APRIL 1994, PP. 259–274

Using Term Rewriting to Verify Software[*]

Sergio Antoy

Portland State University
Department of Computer Science
Portland, OR 97207

John Gannon

University of Maryland
Department of Computer Science
College Park, MD 20742

Abstract

This paper describes a uniform approach to the automation of verification tasks associated with while statements, representation functions for abstract data types, generic program units and abstract base classes. Program units are annotated with equations containing symbols defined by algebraic axioms. An operation's axioms are developed using strategies that guarantee crucial properties such as convergence and sufficient completeness. Sets of axioms are developed by stepwise extensions that preserve these properties. Verifications are performed with the aid of a program that incorporates term rewriting, structural induction, and heuristics based on ideas used in the Boyer-Moore prover. The program provides valuable mechanical assistance: managing inductive arguments and providing hints for necessary lemmas, without which formal proofs would be impossible. The successes and limitations of our approaches are illustrated with examples from each domain.

1 Introduction

Many different methods have been used to annotate software and prove properties about it. Fewer attempts have been made to adapt a single notation to a variety of different annotation tasks and explore the interactions between the types of tasks, properties of the specifications, and demands of verification techniques. In this paper, we apply equational specification and reasoning techniques to verify properties of while statements, abstract data types, generic program units, and derived classes. We present new techniques for computing the weakest preconditions of while statements,

[*]This research has been supported in part by the National Science Foundation grant CCR-8908565 and the Office of Naval Research grants N00014-87-K-0307 and N0014-90-J4091.

annotating abstract base classes from which other classes are derived, and designing algebraic specifications which are convergent and sufficiently complete. In addition, we discuss an experimental tool for partially automating verification activities and report some of our experiences using these techniques and tool.

Rewriting [10, 26] is central to our approach. We use rewriting concepts both for designing small specifications with desirable properties, such as completeness and consistency, and for extending specifications incrementally while preserving these properties. We also use rewriting concepts for proving that programs are correct with respect to their specifications.

Abstraction and factorization reduce the amount of detail that needs to be considered when solving problems. Specifications play a key role in abstraction, hiding details of implementations, and in factoring program components, collecting program units with common semantics rather than just common syntax.

The weakest precondition [11] of a while statement abstracts the particular state transformation induced by a while statement to a class of state transformations which satisfy a particular postcondition [17]. The notion of weakest precondition extends the method proposed in [20] to include termination. Since both termination and verification of programs are unsolvable, it is somewhat surprising that one can compute a first order expression of the weakest precondition of a while statement [8, 32]. This expression, however, involves concepts such as Gödelization or Turing machines, which cannot be reasoned about automatically. We introduce a notation called power functions to describe while statements' state transformations. Power functions are described with algebraic equations, as are the operations which appear in while statements' postconditions. Thus, we may reason about power functions in the same automated way we reason about the operations appearing in program annotations. Power functions also allow us to address incompleteness problems arising in the verification of while statements involving abstract data types [24, 30].

Abstract data types permit program proofs to be factored into two parts: proofs of programs which depend only on abstract properties of objects, and proofs that implementations of types guarantee the abstract properties. In the second type of proof, implementations manipulating concrete objects must satisfy pre and postconditions containing abstract objects. Hoare [21] introduced representation mappings to map concrete objects to their corresponding abstract objects to make such reasoning possible. We show that representation mappings can be cast within the equational framework, allowing us to reap the benefits of equational reasoning and automated term rewriting.

Parameterized subprograms factor similar operations on different objects with the same types, thus reducing the sizes of programs. Generic clauses, like those in Ada, extend the benefits of factoring to program units which manipulate objects with different types. Generic formal type parameters represent classes of types providing a few basic operations (e.g., assignment or equality). Additional generic formal subprogram parameters can be specified to access additional operations. When a generic unit is instantiated, only syntactic discrepancies are reported between the types of the formal and actual generic subprogram parameters. Alphard's designers [31] were among the first to suggest that functions defined with generic type parameters have semantic restrictions that guarantee the functions are properly instantiated. Several research projects are currently investigating how such restrictions should be stated and checked [13, 14, 16]. We use equational reasoning to show that formal parameter specifications denoting properties required of actual parameters are checked when generic program components are instantiated.

Object-oriented programming languages permit new classes to be defined via inheritance. A superclass defines interfaces (and perhaps implementations) for operations, which are inherited by its subclasses. In order to factor the implementation of a common operation in a superclass, each subclass that redefines operations used in implementations of common operations must ensure that its new operations have behaviors which are consistent with those of the respective superclass's operations. That is, each subclass must behave like a subtype of the supertype. Groups of researchers [1, 28, 29] are currently defining subtype relations. We present a method for annotating C++ abstract base classes and other classes which are derived from them. Using equational reasoning, we show that derived classes are subtypes of an abstract base class in a manner similar to that in [19].

In Section 2, we discuss these four classes of verification problems. Although we limit the size of our examples to dimensions suitable for a technical presentation, they are representative of increasingly larger programming problems.

The common denominator for these verification tasks is that we use equations both to annotate each of the program components and to reason about the annotations. In Section 3, we address the problems of both the quality and the expressiveness of specifications on which annotations are based. We motivate the need of structuring specifications as rewrite systems both to ensure crucial properties of specifications and to overcome inherent difficulties of equational reasoning. We present design strategies for extending a specification while preserving its properties as a rewrite system.

In Section 4, we briefly describe an automated tool for formally proving the obligations arising from verification problems. In Section 5, we discuss the use of this tool and informally compare its performance with another automated prover. Section 6 contains our conclusions.

2 Program Annotation and Verification

2.1 While Statements

Power functions [2] are a device to express the weakest precondition of a while statement in a form which is useful for stating and verifying program correctness. We briefly review this technique and show its application in two examples. In a later section we show how to automate the steps of the process.

For any statement w and postcondition R, the weakest precondition $wp(w, R)$ of w with respect to R describes the set of all states S such that when w is activated in a state s in S it terminates in a state r satisfying R [11]. If w is a while statement with condition b and body *stmt*, then

$$wp(w, R) = \exists k : k \geq 0 : H_k(R)$$

where $H_k(R)$ is defined recursively as follows:

$$\begin{aligned} H_0(R) &= \neg b \wedge R \\ H_{k+1}(R) &= b \wedge wp(stmt, H_k(R)) \end{aligned}$$

If the while statement is not defined on some state s, then $wp(w, R)(s)$ is false, since $H_k(R)(s)$ does not hold for any k.

The power function of a function f, whose domain and range are identical, embodies the k-fold composition of f. If $[stmt]$ is the function computed by statement *stmt* and s is a state, the power function pf of $[stmt]$ is defined as

$$pf(k,s) = \begin{cases} s, & \text{if } k = 0; \\ pf(k-1, [stmt](s)), & \text{if } k > 0 \text{ and } [stmt](s) \text{ is defined;} \\ \text{undefined,} & \text{otherwise.} \end{cases} \tag{1}$$

Every function has a unique power function; and the totality, computability, and primitive recursiveness of a function imply similar properties for its power function [2].

Using the notion of power function, we can obtain a first order expression of the weakest precondition of a while statement with respect to any first order postcondition. If $[stmt]$ is a total function and pf is its power function, then

$$H_k(R)(s) = (R(pf(k,s)) \wedge k = \mu i(\neg b(pf(i,s))))$$

The expression $\mu i\, P(i)$ stands for the minimum non-negative integer i, if it exists, such that $P(i)$ holds. More precisely, the second conjunct of the right side is a short hand for $\neg b(pf(k,s)) \wedge \forall i : i < k : b(pf(i,s))$, i.e., k is the least value such that k applications of the $[stmt]$ to the original state produce a state in which b evaluates to false. This yields the following equation

$$wp(w,R)(s) = \exists k : k \geq 0 : (R(pf(k,s)) \wedge k = \mu i(\neg b(pf(i,s)))) \tag{2}$$

The right side requires only pf, the power function of $[stmt]$, which is immediately obtained via equation (1).

Often, we find it convenient to express a power function in terms of other functions that capture higher level abstractions. We show one such example below, where very loosely speaking we say that the power function of a maximum accumulator is the maximum of a sequence. In this case we must ensure the validity of our claim, i.e., we must prove that some function pf is the power function of a given function f. We call this step *validation of pf with respect to f*.

The weakest precondition of a while statement is more manageable when in equation (2) the conjunct $k = \mu i(\neg b(pf(i,s)))$ can be solved with respect to k, i.e., the value of k can be explicitly determined from s. We call this step *minimization of the loop*. Loop minimization is obviously an unsolvable problem since it is more difficult to demonstrate than loop termination. In the following examples we show how to minimize loops and how this operation considerably simplifies the weakest precondition.

Example 1

Consider the following program with while statement w and postcondition R:

```
     $m := a[1];$
     $i := 2;$
$w$: **while** $i \leq n$ **loop**
         **if** $m < a[i]$ **then** $m := a[i];$ **end if**;
         $i := i + 1;$
     **end loop**;
$R$: $\{m = max(a[1..n])\}$
```

where $max(a[1..n])$ is the largest value in the set $\{a[1], \ldots, a[n]\}$.

The function computed by the while statement body ($[stmt]$) returns the program state after examining one component of the array.

$$[stmt](i,a,m) = (i+1, a, max(a[i], m))$$

Its power function (*pf*) returns the program state after examining a slice of the array.

$$pf(k,(i,a,m)) = (i+k, a, max(a[i..i+k-1], m))$$

where $max(a[i..j], m)$ is the largest value in the set $\{a[i], \ldots, a[j], m\}$. We use induction to validate *pf*, i.e., to show that it is indeed the power function of $[stmt]$.

$$\begin{array}{llcl} \text{Base :} & pf(0,(i,a,m)) & = & (i+0, a, max(a[i..i+0-1], m)) = (i,a,m) \\ \text{Ind. :} & pf(k+1,(i,a,m)) & = & (i+(k+1), a, max(a[i..i+(k+1)-1], m)) \\ & & = & ((i+1)+k, a, max(a[i+1..(i+1)+k-1], max(a[i], m))) \\ & & = & pf(k,(i+1, a, max(a[i], m))) \\ & & = & pf(k, [stmt](i,a,m)) \end{array}$$

The minimization of the loop requires us to demonstrate that $\mu k(i+k > n)$ is $n-i+1$. Substituting this expression for k, we can calculate $wp(w,R)$.

$$\begin{array}{rcl} R(pf(n-i+1,(i,a,m))) & = & R(i+n-i+1, a, max(a[i..i+(n-i+1)-1], m)) \\ & = & R(n+1, a, max(a[i..n], m)) \\ & = & (max(a[i..n], m) = max(a[1..n])) \end{array}$$

Although the annotations appear to use familiar, "well-defined" functions such as addition and *max*, we have actually overloaded the function symbol *max*. Assuming all the scalar values are natural numbers, one version of *max* is defined on two naturals, another on an array of naturals, and a third on an array of naturals and a natural. Algebraic axioms permit us to define the relations between symbols that appear in specifications.

$max_0 : nat \times nat \to nat$

$$\begin{array}{lcl} max_0(0,i) & = & i \\ max_0(i,0) & = & i \\ max_0(i+1,j+1) & = & max_0(i,j)+1 \end{array}$$

$max_1 : natarray \times nat \times nat \to nat$

$$\begin{array}{lcl} max_1(a,i,0) & = & a[i] \\ max_1(a,i,k+1) & = & max_0(a[i+k], max_1(a,i,k)) \end{array}$$

$max_2 : natarray \times nat \times nat \times nat \to nat$

$$\begin{array}{lcl} max_2(a,i,0,m) & = & m \\ max_2(a,i,k+1,m) & = & max_0(a[i+k], max_2(a,i,k,m)) \end{array}$$

With these definitions, $wp(w,R)$ can be expressed as:

$$max_2(a,i,n-i+1,m) = max_1(a,1,n)$$

The initializing statements $i := 2; m := a[1]$ transform the above equation into

$$max_2(a,2,n-1,a[1]) = max_1(a,1,n)$$

which can be verified for any a and $n > 0$.

Each of the verification tasks outlined above can be expressed as equations and verified mechanically. These tasks are:

1. Power function validation

Base cases	$add(0, i) = i$
	$max_2(a, i, 0, m) = m$
Inductive cases	$add(k + 1, i) = add(k, i + 1)$
	$max_2(a, i, k + 1, m) = max_2(a, i + 1, k, max_0(a[i], m))$

2. Loop minimization

   $(k < n - i + 1) \Rightarrow (i + k \leq n)$
   $(k = n - i + 1) \Rightarrow (i + k > n)$

3. Loop initialization

   $max_2(a, 2, n - 1, a[1]) = max_1(a, 1, n)$

Example 2

The previous example shows that one may need to define new symbols for the analysis of a loop. This is not a peculiarity of our method. Classic approaches to correctness verification may fail due to the lack of expressiveness of data type specifications [24, 30]. For example, Kamin shows that a program containing the following while statement

```
    $s := s_0$;
    $t := newstack$;
w:  **while** $\neg\ isnewstack(s)$ **loop**
        $t := push(t, top(s))$;
        $s := pop(s)$;
    **end loop**;
R:  $\{t = reverse(s_0)\}$
```

cannot be properly annotated by the lack of expressiveness of the usual theory of type Stack. A similar result appears in [30].

Equation (2) implies that all one needs to properly annotate a loop is the power function of (the functional abstraction of) the loop body. Rather than using equation (1), we chose to formulate the power function of the loop in terms of high-level abstractions. These abstractions capture formally the intuitive concepts that allow a programmer to code the above program.

The repeated execution of the loop body has the effect of chopping off a topmost portion of s, reversing it, and placing it on top of t. The concept of separating a sequence into an initial portion and a remainder generalizes the usual head and tail operations on sequences. We associate the symbols *drop* and *take* with the more general operations and axiomatize them below.

$$
\begin{array}{l}
\mathit{drop} : \mathit{nat} \times \mathit{stack} \to \mathit{stack} \\
\qquad \begin{array}{lcl}
\mathit{drop}(0, s) & = & s \\
\mathit{drop}(i+1, \mathit{newstack}) & = & \mathit{newstack} \\
\mathit{drop}(i+1, \mathit{push}(s, e)) & = & \mathit{drop}(i, s)
\end{array} \\
\mathit{take} : \mathit{nat} \times \mathit{stack} \to \mathit{stack} \\
\qquad \begin{array}{lcl}
\mathit{take}(0, s) & = & \mathit{newstack} \\
\mathit{take}(i+1, \mathit{newstack}) & = & \mathit{newstack} \\
\mathit{take}(i+1, \mathit{push}(s, e)) & = & \mathit{push}(\mathit{take}(i, s), e)
\end{array}
\end{array}
$$

The operation *drop* is denoted pop^* in [24] and is required to make the type stack expressive. *drop* is the power function of *pop*. The operation *take* returns the portion of a stack dropped by *drop*. We formulate the power function of the loop (*pf*) from the functional abstraction of the body ([*stmt*]), exactly as informally stated earlier.

$$
\begin{array}{lcl}
[\mathit{stmt}](s, t) & = & (\mathit{pop}(s), \mathit{push}(t, \mathit{top}(s))) \\
\mathit{pf}(k, (s, t)) & = & (\mathit{drop}(k, s), \mathit{concat}(\mathit{reverse}(\mathit{take}(k, s)), t))
\end{array}
$$

where *concat* and *reverse* are defined as usual.

$$
\begin{array}{l}
\mathit{concat} : \mathit{stack} \times \mathit{stack} \to \mathit{stack} \\
\qquad \begin{array}{lcl}
\mathit{concat}(\mathit{newstack}, s) & = & s \\
\mathit{concat}(\mathit{push}(s, e), t) & = & \mathit{push}(\mathit{concat}(s, t), e)
\end{array} \\
\mathit{reverse} : \mathit{stack} \to \mathit{stack} \\
\qquad \begin{array}{lcl}
\mathit{reverse}(\mathit{newstack}) & = & \mathit{newstack} \\
\mathit{reverse}(\mathit{push}(s, e)) & = & \mathit{concat}(\mathit{reverse}(s), \mathit{push}(\mathit{newstack}, e))
\end{array}
\end{array}
$$

To validate *pf* we prove that

$$
\begin{array}{lcl}
\mathit{drop}(0, s) & = & s \\
\mathit{drop}(i+1, s) & = & \mathit{drop}(i, \mathit{pop}(s)) \\
\mathit{concat}(\mathit{reverse}(\mathit{take}(0, s)), t) & = & t \\
\mathit{concat}(\mathit{reverse}(\mathit{take}(i+1, \mathit{push}(s, e))), t) & = & \mathit{concat}(\mathit{reverse}(\mathit{take}(i, s)), \mathit{push}(t, e))
\end{array}
$$

The last equation is not an instance of the second case of equation (1). It stems from a simple result [2, Th. 5.7] concerning the equivalence of two formulations of power functions, i.e., accumulation vs. recursion.

To minimize the loop we define the operation *size*, which computes the size of a stack, axiomatize *isnewstack*, and verify $\mu k(\mathit{isnewstack}(\mathit{take}(k, s))) = \mathit{size}(s)$.

$$
\begin{array}{l}
\mathit{size} : \mathit{stack} \to \mathit{nat} \\
\qquad \begin{array}{lcl}
\mathit{size}(\mathit{newstack}) & = & 0 \\
\mathit{size}(\mathit{push}(s, e)) & = & \mathit{size}(s) + 1
\end{array} \\
\mathit{isnewstack} : \mathit{stack} \to \mathit{bool} \\
\qquad \begin{array}{lcl}
\mathit{isnewstack}(\mathit{newstack}) & = & \mathit{true} \\
\mathit{isnewstack}(\mathit{push}(s, e)) & = & \mathit{false}
\end{array}
\end{array}
$$

The minimization of the loop is obtained by proving that

$$(k < size(s)) \Rightarrow \neg\, isnewstack(drop(k,s))$$
$$(k = size(s)) \Rightarrow \neg\,\neg\, isnewstack(drop(k,s))$$

The latter is equivalent to $isnewstack(drop(size(s),s))$. Substituting $size(s)$ for k permits us to calculate $wp(w,R)$.

$$\begin{aligned}&R(pf(size(s),s,t)) =\\&R(drop(size(s),s), concat(reverse(take(size(s),s)),t)) =\\&concat(reverse(take(size(s),s)),t) = reverse(s_0)\end{aligned}$$

Initializing t with *newstack* and s with s_0 results in the following wp for the program:

$$concat(reverse(take(size(s_0),s_0)), newstack) = reverse(s_0)$$

which holds for any s_0.

2.2 Data Type Implementations

Modern programming languages provide special constructs to implement user-defined data types. These constructs are specifically designed to hide the representation of a type from its users. Code-level verification techniques, such as those discussed in Examples 1 and 2 are insufficient to address the correctness of an implementation because of the wide gap between the low-level operations performed by the code and the high-level operations described by the operation's interface. For example, decrementing an integer variable may be all it takes to pop a stack. However, verifying that the variable is decremented does not ensure that the code correctly implements the pop operation. We need to show that the code fulfills its obligations to the abstract operations [21].

Example 3

An implementation of the data type stack may represent an instance of the type by a record as follows:

```
subtype index is integer range 1..size;
type data is array (index) of item;
type stack is record
    pntr  : integer range 0..size := 0;
    items : data;
end record;
```

This code fragment belongs to a package with generic arguments *size*, a *positive*, and *item*, a *private type*.

The correctness of an implementation of the type *stack* entails the type's representation mapping [21]. This function, denoted with $\mathcal{A}$ below, maps a concrete instance of *stack*, represented by the above record, to its abstract counterpart.

$$\mathcal{A}(pntr, items) = \begin{cases} newstack, & \text{if } pntr = 0;\\ push(\mathcal{A}(pntr-1, items), items[pntr]), & \text{otherwise.}\end{cases}$$

The implementation of the stack operation *pop* is straightforward.

```
procedure pop(q : in out stack) is
begin
    if $q.pntr = 0$
        then raise underflow;
        else $q.pntr := q.pntr - 1$;
    end if;
end pop;
```

On input a stack *q*, *pop* raises an exception if $q.pntr = 0$, that is, $q.pntr > 0$ is a precondition for the normal termination of the procedure. If the precondition is satisfied, *pop* simply decrements *q.pntr*. The implementation is correct if clients of the stack package, to which the stack representation and the procedure code may be hidden, indeed perceive decrementing *q.pntr* as popping *q*.

The proof method proposed by Hoare reduces the correctness of the implementation to individual obligations of each procedure of the package. Omitting for readability the qualification of *pntr* and *items*, the obligation of the procedure *pop* is

$$pntr > 0 \land \mathcal{A}(pntr, items) = s \; \{\textbf{if } pntr = 0 \; \ldots; \; \textbf{end if};\} \; pntr \geq 0 \land \mathcal{A}(pntr, items) = pop(s)$$

Standard techniques [20] reduce the correctness of the code to the truth of

$$(pntr > 0 \land \mathcal{A}(pntr, items) = s) \Rightarrow \\ ((pntr = 0 \land false) \lor (pntr > 0 \land \mathcal{A}(pntr - 1, items) = pop(s)))$$

where "*false*" in the first disjunct describes the (impossible) initial state that would result in the normal termination of the procedure *pop* when the exception *underflow* is raised. The representation mapping can be defined equationally and the proof obligation can be discharged automatically using the tool discussed in Section 4.

2.3 Instantiations of Generic Program Units

Modularity is an essential feature for the design and implementation of large programs. Generic type and subprogram parameters have been added to statically typed programming languages to avoid duplicating an operation's source code in cases where it manipulates objects only through other operations that are either implicitly defined for its generic formal type parameters or appear as generic formal subprogram parameters. Interconnection errors become more likely and more subtle when such language features are used. Compilers and/or loaders verify only syntactic properties of module interconnections. The verification of (semantic) correctness entails activities similar to those required for the verification of loops and data types discussed earlier, i.e., axiomatizing symbols used for asserting properties or requirements of modules, and proving theorems, expressed by means of these symbols, about the modules.

Example 4

Many computations on sets or sequences of elements are instances of a general paradigm referred to as *accumulation* [4], for example, finding the maximum element, computing the sum of the elements, or counting how many elements have a certain property. These computations can be implemented by a loop whose body processes a new element of the sequence on each iteration. A special variable, whose initial value depends on the computation being performed, "*accumulates*" the result of the computation for the portion of the sequence processed thus far.

Example 1 presented earlier is an instance of accumulation in which the sequence of elements is represented by an array and the process being performed is finding the maximum. In a language supporting generic parameters, the interface of a simple accumulator (in which the types of the elements and the accumulated result are the same) appears as follows:

```
-- specification
generic
   type elem;
   type vector is array(integer range <>) of elem;
   init : elem;
   with function step(a, b : elem) return elem;
function accumulator(v : vector) return elem;

-- body
function accumulator(v : vector) return elem is
   a : elem := init;
begin
   for i in v'first..v'last loop
      a := step(a, v[i]);
   end loop;
   return a;
end accumulator;
```

When a generic subroutine is instantiated (e.g., with actual parameters *natural*, *nat_vect*, 0, and *max*, as shown below) discrepancies may be detected between the types of the generic subprogram parameters and the types of the actual objects bound to them.

```
procedure main is
   ...
   function max_array is new
      accumulator(elem ⇒ natural, vector ⇒ nat_vect, init ⇒ 0, step ⇒ max);
   ...
```

Unfortunately, only syntactic discrepancies are reported. Some implementations of an accumulator may rely on semantic properties which do not hold for all bindings, but cannot be detected by the compiler.

For example, certain accumulations can be performed in parallel. In the simplest form, a parallel implementation of an accumulator may simultaneously activate two tasks. Each task is an accumulator operating on half of the input array and feeding its results to the function *step* which returns the desired value. To improve the implementation's efficiency, we can use a tree-like cascade of tasks each executing a single invocation of *step* in parallel. However, the parallel implementation of the accumulator assumes that the function bound to the generic parameter *step* is associative and that the element bound to the parameter *init* is its left identity i.e., $(elem; step, init)$ is a monoid.

This result can be established in the following manner. Let $e_1, e_2, \ldots$ be the sequence of values processed by the accumulator, and $\mathcal{A}$ the function defined by

$$\mathcal{A}(e_i, \ldots, e_j) = \begin{cases} \mathit{init}, & \text{if } i > j; \\ \mathit{step}(\mathcal{A}(e_i, \ldots e_{j-1}), e_j), & \text{otherwise.} \end{cases}$$

With the techniques described in Section 2.1 we can prove that $\mathcal{A}$ is the function computed by the code of *accumulator*. If for all k such that $i \leq k \leq j$, the following equation holds

$$\mathcal{A}(e_i, \ldots, e_j) = \mathit{step}(\mathcal{A}(e_i, \ldots, e_k), \mathcal{A}(e_{k+1}, \ldots, e_j)) \tag{3}$$

we can implement our accumulator in parallel as described above. It is easy to show that equation (3) holds when *elem* is a monoid.

Algebraic notation can be used to specify properties of generic subroutine parameters that can be verified from the specifications of the actual parameters. Such restrictions can be made explicit by writing them as conditions and including them with the text of the specification of *accumulator*.

$$\mathit{step}(\mathit{step}(x, y), z) = \mathit{step}(x, \mathit{step}(y, z))$$
$$\mathit{step}(\mathit{init}, x) = x$$

When the function *accumulator* is instantiated with actual arguments replacing the formal parameters, the identifiers in the axioms of the actuals can be replaced by the names of the formals and the specification of the actual arguments can be used to prove these conditions. For example, it is easy to verify these conditions for the operation max_0, the maximum of two natural numbers, specified in Example 1. Likewise, the instantiation requirement holds for both addition and multiplication, but not for exponentiation. Thus, exponentiation cannot be legally bound to the generic parameter *step*.

2.4 Inheritance

Object-oriented programming languages permit the definition of new classes via inheritance. A subclass inherits data representations and operations from a superclass and may add or redefine these components. We use algebraic equations to specify both the behavior of classes and to verify that a subclass relation is also a subtype relation.

Example 5

In the following example, `Shape` is an abstract class; it can serve as a superclass for another class but no objects of type `Shape` may be created.

```
class Shape {
  public:
    virtual Point center () const {
      return Point ((left() + right()) / 2, (top() + bottom()) / 2); };
    virtual void  move (const Point & P) = 0;
    void recenter (const Point& p) { move (p-center()); };
    virtual double top ()    const = 0;
    virtual double bottom () const = 0;
    virtual double left ()   const = 0;
    virtual double right ()  const = 0;
};
```

An abstract class is used to define interfaces for operations which manipulate objects created by its subclasses. For example, `recenter` moves an object to a new position.

`Circle` is declared as a subclass of `Shape`, redefining the latter's `center` operation with a more efficient version of its own and providing definitions for those operations which are pure virtual functions in `Shape` (i.e., `move`, `top`, etc.).

```
class Circle : public Shape {
  public:
    Circle (const Point & C, const double & R) : _center (C) { radius (R); };
    inline void radius (const double & R) { assert (R>=0); _radius = R; };
    inline double radius () const { return _radius; };
    inline void center (const Point & C) { _center = C; };
    inline Point center () const { return _center; };
    inline void  move (const Point & P) { _center.move(P); };
    double top ()    const { return (_center.y() + _radius); };
    double bottom () const { return (_center.y() - _radius); };
    double left ()   const { return (_center.x() - _radius); };
    double right ()  const { return (_center.x() + _radius); };
  private:
    Point _center;
    double _radius;
};
```

When it is passed a reference to a `Circle` object, `recenter` invokes `Circle`'s `center` and `move` operations.

```
Point   p1(10,10), p2(5,5);
Circle  c(p1,20);
...
c.recenter(p2);
```

We can use algebraic specifications to define meanings for `Shape`'s operations. The first argument of an abstract operation f modeling a corresponding concrete operation `f` is the class instance to which `f` belongs. For example, referring to the above program fragment, $recenter(c, p2)$ is the abstract counterpart of `c.recenter(p2)`.

We do not specify an abstract class, such as `Shape`, by means of a sort. Rather, we describe relationships between the class' defined operations. The completeness of our specification is a critical issue. Heuristically, we consider each pair, triple, etc. of member functions of `Shape` and capture their mutual dependencies, if any, by algebraic equations. We remove obviously redundant equations.

$$\begin{array}{l}
\mathit{center}(\mathit{move}(S,P)) = \mathit{center}(S) + P \\
\mathit{top}(\mathit{move}(S,\mathit{point}(X,Y))) = \mathit{top}(S) + Y \\
\mathit{bottom}(\mathit{move}(S,\mathit{point}(X,Y))) = \mathit{bottom}(S) + Y \\
\mathit{left}(\mathit{move}(S,\mathit{point}(X,Y))) = \mathit{left}(S) + X \\
\mathit{right}(\mathit{move}(S,\mathit{point}(X,Y))) = \mathit{right}(S) + X \\
\mathit{center}(S) = \mathit{point}((\mathit{left}(S) + \mathit{right}(S))/2, (\mathit{top}(S) + \mathit{bottom}(S))/2) \\
\mathit{recenter}(S,P) = \mathit{move}(S, P - \mathit{center}(S)) \\
\mathit{left}(S) \leq \mathit{right}(S) \\
\mathit{bottom}(S) \leq \mathit{top}(S)
\end{array}$$

These specifications define the meanings of operations which manipulate objects of type `Shape`. Using these specifications we may prove the correctness of the implementation of member functions which are not pure virtual, by assuming the correctness of the "future" implementation of the member which are pure virtual. As discussed earlier, the verification condition is

$$\mathcal{A}(\mathit{this}) = s \wedge p = p' \; \{ \texttt{move (p-center());} \} \; \mathcal{A}(\mathit{this}) = \mathit{recenter}(s,p) \wedge p = p'$$

where the conjunct $p = p'$ ensures that the argument of `recenter` remains constant.

The proof of the implementation of `recenter` relies on the pre- and post-conditions of `Shape`'s `center` and `move` operations. For such proofs to hold when `recenter` is passed an object whose type is derived from `Shape`, the object's type must be a subtype, not merely a subclasses, of type `Shape`. To show this, we must demonstrate that the relationships among the operations of `Shape` hold for the operations and instances of `Circle`.

The specifications of `Circle` (shown below) differ from those of `Shape`, since the latter is a classic abstract data type, rather than an abstract class in the C++ sense. The first condition is the class invariant. It ensures that every `Circle` has a non-negative radius.

$$\begin{array}{l}
S = \mathit{circle}(C,R) \Rightarrow R \geq 0 \\
R \geq 0 \Rightarrow \mathit{radius}(\mathit{circle}(C,Q),R) = \mathit{circle}(C,R) \\
\mathit{radius}(\mathit{circle}(Q,R)) = R \\
\mathit{center}(\mathit{circle}(C,R)) = C \\
\mathit{move}(\mathit{circle}(C,R),P) = \mathit{circle}(C+P,R) \\
\mathit{top}(\mathit{circle}(\mathit{point}(X,Y),R)) = Y + R \\
\mathit{bottom}(\mathit{circle}(\mathit{point}(X,Y),R)) = Y - R \\
\mathit{left}(\mathit{circle}(\mathit{point}(X,Y),R)) = X - R \\
\mathit{right}(\mathit{circle}(\mathit{point}(X,Y),R)) = X + R
\end{array}$$

`Circle`'s operations can be annotated as usual [21], although the standard proof techniques for imperative languages [11, 20] may fall short to prove object-oriented code.

We are concerned with a different problem here, that is, we want to prove that `Circle` is a subtype of `Shape`. For this task we verify that the annotations of `Shape` hold for every instance of `Circle`. This activity is similar to proving that `Circle` implements `Shape` with the technique proposed in [19], with minor a difference—a `Circle` *is* a `Shape`, thus, no representation function or equality interpretation is involved in the proofs.

Most of these proofs are easily formulated as problems for our theorem prover and completed automatically.

3 Designing Specifications for Annotations

The problems discussed in the previous sections are formulated and resolved using first order formulas. These formulas involve the symbols of a specification whose atomic components are equations. In this section we discuss how we design both our equations and specifications. Our goal is to produce equations and specifications that are easy to process automatically. The processing is not limited to proving the formula expressing the correctness of a piece of software, but also includes analyzing the specifications to determine that they satisfy properties whose absence is often a sign of flaws.

A major obstacle to automation is the declarative nature of equations. Changing equations into rewrite rules makes a specification more operational and simplifies the problem.

3.1 Term Rewriting

The unrestricted freedom, provided by equational reasoning, of replacing a term with an equal term leads to a combinatorial explosion of possibilities which are hard to manage by a prover, whether automated or human. An equation $t_1 = t_2$ can be "oriented" yielding a rewrite rule $t_1 \to t_2$. This rewrite rule still defines the equality of t_1 and t_2. It allows the replacement of an instance of t_1 with the corresponding instance of t_2, but forbids replacement in the opposite direction. Orienting equations transforms an algebraic specification into a *term rewriting system* [10, 26].

There are two crucial properties that must be achieved when equations are oriented. Two terms provably equal by equational reasoning, should have a common reduct, i.e., a third term to which both can be rewritten. This property is referred to as *confluence* or *Church-Rosser.* In addition, it should not be possible to rewrite a term forever, in particular there should be no circular rewrites. This property is referred to as *termination* or *Noetherianity.* A system with both properties is *canonical* or *complete* or *convergent.* The Knuth-Bendix completion procedure [27] attempts to transform an equational specification into a complete rewrite system. The termination of the procedure cannot be guaranteed and its execution may require human intervention. The difficulty stems from the undecidability of whether or not a rewrite system is canonical [9, 22].

For this reason, we do not attempt to convert an equational specification in the corresponding complete rewrite system. Rather, we ask specifiers to structure their specifications as rewrite systems with the above characteristics. The task is eased considerably by two strategies used in designing a specification. The technique also ensures other properties, such as sufficient completeness, which we deem essential in our framework.

3.2 Sufficient Completeness of Constructor-Based Systems

To apply our technique we consider only *constructor-based* systems, i.e., we partition the signature symbols into *constructors* and *defined operations.* The constructors of a type T generate all the data instances or *values* of T which are represented by terms, called *normal forms*, that cannot be reduced. Terms containing defined operations represent computations. For example, the constructors of the natural numbers are 0 and *successor* (denoted by the postfix "+1" in the examples). The constructors of the type *stack* discussed in Example 2 are *newstack* and *push*, since any stack is either empty or is obtainable by pushing some element on some other stack. *Concat* and *reverse* are examples of defined operations.

Considering constructor-based systems raises the problem of sufficient completeness, yet another undecidable property [25]. For the specification of type T to be *sufficiently complete*, it must assign a value to each term of type T [18]. If the specification is structured as a constructor-based rewrite system, sufficient completeness is equivalent to the property that normal forms are constructor terms. If left sides of axioms have defined operations as their outermost operators and constructor terms as arguments, we can state necessary and sufficient conditions for the sufficient completeness of a specification.

A *constructor enumeration* [7] is a set, C, of tuples of constructor terms such that substituting constructor terms for variables in the tuples of C exhaustively and unambiguously generates the set of all the tuples of constructor terms. The set of tuples of arguments of a defined operation should be a constructor enumeration. For example, the set of tuples of arguments of *drop*, discussed in Example 2 and shown below

$$C = \{\langle 0, s\rangle, \langle i+1, newstack\rangle, \langle i+1, push(s,e)\rangle\}$$

is a constructor enumeration of $\langle nat, stack\rangle$, since every pair $\langle x, y\rangle$, with x natural and y stack is an instance of one and only one element of C.

The set of tuples of arguments of the operation max_0 discussed in Example 1 is not a constructor enumeration, since $\langle 0,0\rangle$ is an instance of both $\langle 0,i\rangle$ and $\langle i,0\rangle$. The second axiom of max_0 should have been

$$max_0(i+1, 0) = i+1$$

Although the difference does not affect the specification, the latter axiomatization removes a (trivial) ambiguity. Note that if the right side of the second axiom of max_0 were defined as $i+1$, rather than i, the specification would be inconsistent since $0=1$ would be a consequence of the axioms.

An operation is *overspecified* when two rules can be used to rewrite the same combination of arguments. It is *underspecified* when no rule can be used to rewrite some combination of arguments. Overspecification can be detected by a superposition algorithm [27] which uses unification to detect overlapping. Underspecification is a natural condition for some operations, although it creates non-negligible problems. It can be systematically avoided, for example, in the framework of order-sorted specifications [15]. Underspecification can be detected by an algorithm informally described below [23]. Operations that are not underspecified are called *completely defined.*

Huet and Hullot devised an algorithm to detect incompletely defined operations. This algorithm assembles the arguments of the axioms into tuples and the terms in the i^{th} position of each tuple are checked to see that they include a variable or "an instance of each constructor." For each constructor c, tuples of remaining arguments with constructor c or a variable in position i are formed and recursively tested. A rigorous description appears in [23].

By way of example, we execute this algorithm with the set C as input. For constructor 0 in position 1, tuple 1 is considered. The set of tuples of remaining arguments, $\{\langle s\rangle\}$, is trivially complete. For constructor *successor* in position 1, tuples 2 and 3 are considered. The set of tuples of remaining arguments is $\{\langle newstack\rangle, \langle push(s,e)\rangle\}$. It contains an instance of each constructor of *stack* in position 1. The completeness of each recursive problem, $\{\langle\rangle\}$ from *newstack* and $\{\langle s,e\rangle\}$ from *push*, is obvious.

To ensure the confluence of a constructor-based specification it is sufficient to avoid overspecification. To ensure sufficient completeness it is necessary, but not sufficient, to avoid underspecification. If all operations are completely defined and terminating, then the specification is sufficiently complete. In fact, every term has a normal form which obviously contains only constructor symbols because any term containing a defined operation is reducible. Underspecification and overspecification are easily checked syntactic properties. However, the termination of a rewrite system is undecidable [9]. In the next section, we discuss syntactic properties sufficient to ensure termination and show how to obtain them through our design strategies.

3.3 Design Strategies for Axioms

Confluence and sufficient completeness are undecidable, although essential, properties of a specification. Lack of confluence implies that some computation is ambiguously specified. Lack of sufficient completeness implies that some computation is unspecified. We regard both conditions as serious flaws of a specification. We describe two design strategies for generating confluent and sufficiently complete specifications.

The *binary choice strategy* is an interactive, iterative, non-deterministic procedure that through a sequence of binary decisions generates the left sides of the axioms of a defined operation. We used the symbol "$\Box$", called *place*, as a placeholder for a decision. Let f be an operation of type $s_1, \ldots, s_k \to s$. Consider the template $f(\Box, \ldots, \Box)$, where the i^{th} place has sort s_i. To get a rule's left side we must replace each place of a template with either a variable or with a constructor term of the appropriate sort. In forming the left sides, we neither want to forget some combination of arguments, nor include other combinations twice. That is, we want to avoid both underspecification and overspecification. This is equivalent to forming a constructor enumeration.

We achieve our goal by selecting a place in a template and chosing one of two options: *"variable"* or *"inductive."* The choice *variable* replaces the selected place with a fresh variable. The choice *inductive* for a place of sort s_i splits the corresponding template in several new templates, one for each constructor c of sort s_i. Each new template replaces the selected place with $c(\Box, \ldots, \Box)$, where there are as many places as the arity of c. A formal description of the strategy appears in [3]. We apply the strategy for designing the (left sides of the) rules of the operation *drop* discussed in Example 2. The initial template is

$drop(\Box, \Box)$

We chose *inductive* for the first place. Since the sort of this place is *natural*, we split the template into two new templates, one associated with 0 and the other with *successor*

$drop(0, \Box)$
$drop(\Box + 1, \Box)$

We now chose *variable* for the remaining place of the first template and *variable* again for the first place of the second template to obtain:

$drop(0, s)$
$drop(i + 1, \Box)$

We chose *inductive* for last remaining place. Since the sort of this place is *stack*, we again split the template in two new templates, one associated with *newstack* and the other with *push*. We obtain:

$drop(0, s)$
$drop(i + 1, newstack)$
$drop(i + 1, push(\Box, \Box))$

For each remaining choice, *variable* is selected, completing the rules' left sides.

We now describe the second strategy, which ensures termination. The *recursive reduction* of a term t is the term obtained by "stripping" t of its recursive constructors. A constructor of sort s is called *recursive* if it has some argument of sort s. For example, *successor* and *push* are recursive constructors. "Stripping" a term $c(t_1, \ldots, t_k)$, where c is a derived operation and t_i is a recursive constructor, removes the outermost application of the constructor from t_i. The stripping process is recursively applied throughout the term. A formal description of the *recursive reduction* function appears in [3]. We show its application in examples.

For reasons that will become clear shortly, we are interested in computing the recursive reduction of the left side of a rewrite rule for use in the corresponding right side. The symbol "\$" in the right side of a rule denotes the recursive reduction of the rule's left side. With this convention, the last axiom of *drop* is written

$$\mathit{drop}(i+1, \mathit{push}(s, e)) = \$$$

since the recursive reduction of the left side is $\mathit{drop}(i, s)$. We obtain it by replacing $i+1$ with i since i is the recursive argument of *successor*, and by replacing $\mathit{push}(s, e)$ with s since s is the recursive argument of *push*. When a constructor has several recursive arguments the recursive reduction requires an explicit indication of the selected argument. We may also specify a partial, rather than complete, "stripping" of the recursive constructors.

The *recursive reduction strategy* consists in defining the right sides of rules using only functional composition of symbols of a terminating term rewriting system and the recursive reduction of the corresponding left sides. If a specification is designed using the binary choice and the recursive reduction strategies, then it is canonical and sufficiently complete [3].

3.4 Design Strategies for Specifications

The above strategies lead naturally to the design approach called stepwise specification by extensions [12]. Given a specification S_i, a step extends the specification by adding some operations and yielding a new specification S_{i+1}. S_{i+1} is a *complete* and *consistent* extension of S_i [12], if every data element of S_{i+1} was already in S_i and distinct elements of S_i remain distinct in S_{i+1}. Furthermore, if S_i is canonical and sufficiently complete, then so is S_{i+1} [3].

We clarify these concepts by showing the steps yielding the specification of Example 2. Our initial specification, S_0, consists of the sorts *boolean*, *natural* and *stack* with their constructors only, i.e., *true*, *false*, 0, *successor*, *newstack*, and *push*. Since there are no rewrite rules, i.e., the constructors are free, the canonicity and sufficient completeness of S_0 are trivially established. Now we extend S_0 with the operation *concat* obtaining S_1.

$$\begin{array}{lcl}
\mathit{concat}(\mathit{newstack}, s) & = & s \\
\mathit{concat}(\mathit{push}(s, e), t) & = & \mathit{push}(\$, e)
\end{array}$$

The recursive reduction of the left side is $\mathit{concat}(s, t)$. Since we designed the *concat* axioms using the binary choice and recursive reduction strategies, S_1 is a complete and consistent extension of S_0 and is a canonical and sufficiently complete specification. During this step we may also extend S_0 with *drop*, *take*, *size*, and *isnewstack*. However, we cannot extend S_0 with *reverse* because the right side of one axiom of *reverse* contains *concat*. We must first establish the properties of the specification containing *concat*. Hence, a separate step is necessary. Then, we extend S_1 with the operation *reverse* obtaining S_2.

$$\begin{array}{lcl} reverse(newstack) & = & newstack \\ reverse(push(s,e)) & = & concat(\$, push(newstack, e)) \end{array}$$

Our strategies together with the stepwise approach ensure again that S_2 is a complete and consistent extension of S_1 and is a canonical and sufficiently complete specification. All the specifications presented in this note are designed in this manner.

The binary choice strategy force us to construct left sides of plausible axioms for which we do not want to define a right side. We complete the definition of these "axioms" by placing the distinguished symbol "?" in the right side. Other specification languages follow an equivalent approach to control incompleteness. For example, Larch would declare as "exempt" any term appearing as left side of one of the axiom we single out with "?" Our strategies can be used also in the presence of non-free constructors. Some properties of specifications with non-free constructors, such as confluence, are no longer automatically guaranteed, but they can be checked more easily than when no strategies at all are used in the design of the specification [3].

4 Proving Theorems About Annotations

The examples in the previous section contain many small theorems that need to be proved. Automating these proofs makes them easier to carry out and less prone to error. In this section we report our experience with this task.

4.1 Induction

Many equations, e.g., $X + Y = Y + X$, cannot be proved by rewriting, i.e., using only equational reasoning. Such equations can be proved via structural induction [6] or data type induction [19]. Inductive variables of type T are replaced by terms determined by T's constructors and inductive hypotheses are established. If F is a formula to be proved, v is the inductive variable, and s is the type of v, our induction proofs are carried out in the following manner. For every constructor c of type $s_1 \times \ldots \times s_n \rightarrow s$ for $n \geq 0$, we prove $F[c(v_1, \ldots, v_n)/v]$, where v_i, $1 \leq i \leq n$, is a distinct Skolem constant; and if $s_i = s$, then $F[v_i/v]$ is an inductive hypothesis.

4.2 An Automated Theorem Prover

We have implemented a prototype theorem prover incorporating many concepts from the Boyer-Moore Theorem Prover [5]. However, except for built-in knowledge of term equality and data type induction, the knowledge in the theorem prover is supplied by specifications. Our theorem prover checks that each function is completely defined by executing Huet's inductive definition. During this check it identifies as "inductive" those arguments filled by instances of constructors. The discovery of inductive arguments allows the prover to generate automatically the theorems which constitute the cases of a proof by induction.

The theorem prover executes four basic actions: *reduce*, *fertilize*, *generalize*, and *induct*. *Reduce* applies a rewrite rule to the formula being proved. *Fertilize* is responsible for "using" an inductive hypothesis (i.e., replacing a subterm in the current formula with an equivalent term from an inductive hypothesis). *Generalize* tries to replace some non-variable subterm common to both sides of the formula with a fresh variable. *Induct* selects an inductive variable and generates new equations. An induction variable is chosen from the set of the inductive arguments by heuristics which include popularity [5] and seniority.

The theorem prover computes a boolean recursive function, called *prove*, whose input is an equation and whose output is *true* if and only if the equation has been proved. Axioms and lemmas of the specification are accessed as global data. Proofs of theorems are generated as side effects of computations of *prove*. Users may override the automatic choices, made by the prover, for inductive variables and generalizations. A technique discussed later allows users to use case analyses in proofs.

```
function $prove(E)$ is
    begin
        if $E$ has the form $x = x$, for some term $x$, then return $true$; end if;
        if $E$ can be reduced, then return $prove(reduce(E))$; end if;
        if $E$ can be fertilized, then return $prove(fertilize(E))$; end if;
        $E' := generalize(E)$;
        if $E'$ contains an inductive variable, then
            $E_1, \ldots, E_n := induct(E')$;
            return $prove(E_1)$ andalso $\ldots$ andalso $prove(E_n)$;
        end if;
        return $false$;
    end $prove$;
```

An attempt to prove a theorem may exhaust the available resources, since induction may generate an infinite sequence of formulas to be proved. However, the termination property of the rewrite system guarantees that an equation cannot be reduced forever and the elimination of previously used inductive hypotheses [5] guarantees that an equation cannot be fertilized forever.

5 Experience Proving Theorems

All the proofs discussed in the previous sections have been completely generated with our theorem prover, except for two proofs of inheritance properties. Many proofs were produced automatically by the prover. Others were generated only after we supplied additional lemmas, independently proved using the theorem prover.

The validation proofs for power functions for both while statement examples were all done automatically, some with just term rewriting and the others with rewriting and induction. The minimization proofs were slightly more challenging. Those for the array example required three simple lemmas (e.g., $X + 1 > Y = X \geq Y$) to be proved and added to the set of axioms before the theorem prover could finish the proofs. The lemmas were suggested by the similarity of terms on opposite sides of the equations generated by the theorem prover. Generalization had to be inhibited to obtain the minimization proofs for the stack example, as explained below. Relationships between different Skolem constants inserted at the same time may be lost when generalization replaces terms containing these constants with new constants. In attempting to verify

$$(k < size(s)) \Rightarrow \neg isnewstack(drop(k, s))$$

we generate the equation

$$((1 < size(A1)) \Rightarrow \neg isnewstack(drop(1, A1))) = ((0 < size(A1)) \Rightarrow \neg isnewstack(A1))$$

Generalization replaces $size(A1)$ with a new Skolem constant $B2$ and starts to verify the lemma

$$((1 < B2) \Rightarrow \neg isnewstack(drop(1, A1))) = ((0 < B2) \Rightarrow \neg isnewstack(A1))$$

The relation between $A1$ and $B2$, lost by the generalization, is crucial to the validity of the theorem. In nested inductions, for $B2 = 1$ and $A1 = newstack$, this equation is rewritten to

$$true = false$$

and the proof attempt fails. Simply inhibiting generalization in this case solves the problem and the theorem is proved with just rewriting and induction.

However, generalization is essential to other proofs. We illustrate this by an example, which also shows how we discover the lemmas that make some proofs possible or simpler. The proof of the total correctness of Example 2 requires verifying

$$concat(reverse(take(size(s), s)), newstack) = reverse(s)$$

During an induction on s, the formula to be proved becomes

$$\begin{aligned} &concat(concat(reverse(take(size(A1), A1)), push(newstack, A2)), newstack) \\ &\quad = concat(concat(reverse(take(size(A1), A1)), newstack), push(newstack, A2)) \end{aligned}$$

The prover simultaneously generalizes $reverse(take(size(A1), A1))$ and $push(newstack, A2)$ and attempts to prove

$$concat(concat(A6, A7), newstack) = concat(concat(A6, newstack), A7) \tag{4}$$

The proof is easily completed by nested inductions on $A6$ and $A7$. Without generalization, the proof continues by induction on $A1$, but the inductive hypothesis is not strong enough to complete the proof. The prover keeps generating new inductions on formulas with increasing complexities until the available resources are exhausted.

We regard generalized formulas as lemmas. Often we are able to further generalize the lemmas suggested by the prover. For example, equation (4) suggests that *newstack* might be a right identity of *concat*. Thus we prove

$$concat(s, newstack) = s$$

and use it as a lemma in the original proof. This immediately reduced the original formula to

$$reverse(take(size(s), s)) = reverse(s)$$

The presence of a leading *reverse* on each side of the equation suggests that the equality may depend on the arguments only. Thus we attempt to prove

$$take(size(s), s) = s$$

The proof succeeds and we use this result as a lemma in the original proof. With these two lemmas the original proof becomes trivial. Both lemmas are obtained by "removing context." Generalization hypothesizes that the truth of an equation does not depend on certain *internal* specific portions of each side. The latter example hypothesizes that the truth of an equation does not depend on certain *external* specific portions each side, i.e., $reverse(\ldots)$.

A proof of the conditions required for the semantic correctness of the generic instantiations of *accumulator* was attempted for the operations max_0 (see Example 1), addition, multiplication, and exponentiation. The theorems relative to the first three instantiations were all proved automatically. However, the attempt to prove the associativity of exponentiation fails. The prover attempts to verify $(x^y)^z = x^{(y^z)}$ by a nested induction. For $z = 0$ and $x = 0$ the equation is reduced to $1 = 0$ and the prover halts with a message that it failed for this case. Thus, only the first three instantiations of the generic accumulator are semantically correct. Interestingly, the fact that equation (3) holds when *step* is associative and *init* is its left identity was proved automatically by our prover too.

The reversal of a stack discussed in Example 2 can be parallelized by a divide-and-conquer technique similar to that discussed for accumulation. This program is an instance of a more complex case of accumulation, in which the type of the result of the accumulation differs from the type of the of elements of the accumulated sequence. We may exploit parallelism if we assume that a stack is dynamically allocated in "chunks." Each chunk consists of a fixed-size array-like group of contiguous memory locations, which are addressed by an index. Chunks are allocated on demand and do not necessarily occupy contiguous locations of memory, rather are threaded together by pointers as in a linked list. We reverse a stack in parallel only when the stack consists of several chunks. In this case, we split the stack in two non-empty portions, say x and y, of linked together whole chunks. We assume that the bottommost chunk of x points to the topmost chunk of y. We reverse this link and recursively reverse x and y. When a portion of a stack consists of a single chunk, we only have to swap the content of the chunk's memory locations to achieve reversal. The correctness of this parallel implementation of a stack reversal relies on the equation

$$reverse(concat(x, y)) = concat(reverse(y), reverse(x))$$

where *reverse* and *concat* were defined in Example 2. Operationally, *concat* stands for the operation linking together two portions of a stack represented by its arguments. *Reverse* is overloaded: on multi-chunk stacks it partitions and recurs whereas on single-chunk stacks it swaps. The proof of the equation entails only the mutual relationships between these symbols, not their implementations. Thus, the representation "in chunks" of the type *stack* is not an issue of the proof. Obviously, in a comprehensive proof of correctness the differences between the two computations associated to the symbol *reverse* must be accounted for, e.g., as discussed in Section 2.2.

Our prover proves the above equation without human help. Interestingly, during the proof, which is by induction on x, the prover automatically generates and proves an independent lemma for each case of the induction. One lemma states that *newstack* is a right identity of *concat*, the other that *concat* is associative.

In Section 2.4, we presented annotations involving the C^{++} predefined type *double*. We did not axiomatize this type by means of an algebraic inductive specification, and consequently we could not use our prover for theorems relying on intrinsic properties of this type. However, all the proofs in this section, except $left(S) \leq right(S)$ and $bottom(S) \leq top(S)$, were easily proved when we provided some simple unproved lemmas, such the commutativity of "+" for *double*.

5.1 An Informal Comparison

The need to supply guidance to an automatic prover is not a peculiarity of our implementation. For example, the Larch Theorem Prover (LP) [13] is designed to be a proof checker as well as an automated prover. We consider this a approach very sensible.

The guidance required by our prover is in the form of lemmas. Lemmas simplify proofs and improve understanding by "removing context." In our case they also overcome the lack of certain proof tactics and of a friendly interface in our implementation.

We briefly compare how LP and our prover prove two sample theorems proposed in [13]. The proofs involve the types *linear container* and *priority queue*, whose axioms are shown below, a *total order*, and the type *boolean* with standard operators.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{isempty : queue \rightarrow boolean} \\
\quad isempty(new) & = & true \\
\quad isempty(insert(C,E)) & = & false \\
\multicolumn{3}{l}{member : queue \rightarrow boolean} \\
\quad member(new,E) & = & false \\
\quad member(insert(C,E),F) & = & ((E = F) \vee member(C,F)) \\
\multicolumn{3}{l}{next : queue \rightarrow nat} \\
\quad next(new) & = & ? \\
\quad next(insert(Q,C)) & = & \textbf{if } isempty(Q) \textbf{ then } C \\
 & & \textbf{else if } next(Q) < C \textbf{ then } next(Q) \\
 & & \textbf{else } C
\end{array}
$$

The "?" symbol in the axioms of *next* corresponds to the Larch declaration *"exempt"* for $next(new)$. LP was used to prove the theorems:

$$
\begin{array}{lcl}
isempty(C) & \Rightarrow & \neg member(C,E) \\
member(Q,E) & \Rightarrow & \neg(E < next(Q))
\end{array}
$$

The first theorem was proved after an inductive variable (C) was explicitly picked. The second theorem required more intervention after rewriting produced:

$$
\begin{array}{l}
((E = V) \vee member(Q,E)) \Rightarrow \\
\quad \neg(E < (\textbf{if } isempty(Q) \textbf{ then } V \textbf{ else if } V < next(Q) \textbf{ then } V \textbf{ else } next(Q)))
\end{array}
$$

LP was given a series of commands to divide the proof into cases and apply critical-pairs completions. The case $isempty(Q)$ required a critical-pairs completion. The case $\neg isempty(Q)$ required case analysis on the truth $V < next(Q)$. Each case was further subdivided based on the truth of $E = V$. For the case $\neg(V < next(Q))$ and $\neg(E = V)$ another critical-pairs completion was requested.

Our prover also verified both these theorems, the first automatically and the second after we added a few lemmas to the axioms. Since our prover's only knowledge derives from axioms, we do not treat conditional expressions or implications in any special manner. They are represented by means of user-defined operations. For example, implication is defined by the following axioms.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\Rightarrow : boolean \times boolean \rightarrow boolean} \\
\quad true \Rightarrow X & = & X \\
\quad false \Rightarrow X & = & true
\end{array}
$$

Thus we often need lemmas to manipulate these functions.

Since our prover handles only equations, we formulate the second theorem as

$$(\mathit{member}(A0, A1) \Rightarrow \neg(A1 < \mathit{next}(A0))) = \mathit{true}$$

The prover automatically chooses $A0$ as the inductive variable. The base case, $A0 = \mathit{new}$, is trivially proved by rewriting. The inductive case, $A0 = \mathit{insert}(A2, A3)$, gives $(\mathit{member}(A2, A1) \Rightarrow \neg(A1 < \mathit{next}(A2))) = \mathit{true}$ as an inductive hypothesis and reduces the left side to

$$((A3 = A1) \vee \mathit{member}(A2, A1)) \Rightarrow \xi$$

where ξ is a large nested conditional expression. We use two "standard" lemmas to simplify ξ. One distributes "<" with respect to conditionals, i.e., we replace instances of $x <$ **if** b **then** y **else** z with **if** b **then** $x < y$ **else** $x < z$. This transformation allows the use of properties of the ordering relation "<" such as irreflexivity and the "implied equation" [13, Fig. 9]. The other lemma splits implications whose antecedent is a disjunction, i.e., we replace instances of $x \vee y \Rightarrow z$ with $(x \Rightarrow z) \wedge (y \Rightarrow z)$. This transformation allows the use of each antecedent independently. By applications of these lemmas, left side of the equation to be proved becomes

$$((A3 = A1) \Rightarrow \xi') \wedge (\mathit{member}(A2, A1) \Rightarrow \xi')$$

Now we enable the use of the antecedent of each conjunct of the formula by means of two "specific" lemmas. This approach is directly inspired by [5]. One lemma replaces instances of $(x = y) \Rightarrow P(y)$ with $(x \neq y) \vee P(x)$, the other lemma replaces instances $Q \Rightarrow P(y)$ with $\neg Q \vee P(x)$, if $Q \Rightarrow (x = y)$ holds. The specific instance of the latter is $\mathit{member}(A2, A1)$ => $(\mathit{isempty}(A2) = \mathit{false})$, i.e., the contrapositive form of the first theorem proved for this problem.

After several inference steps, the left side is reduced to:

$$\neg\mathit{member}(A2, A1) \vee \neg(((\mathit{next}(A2) < A3) \wedge (A1 < \mathit{next}(A2))) \vee \\ (\neg(\mathit{next}(A2) < A3) \wedge (A1 < A3)))$$

Now, we continue by cases on $\mathit{next}(A2) < A3$ because this expression and its negation appear in the formula. Although, our prover lacks a "proof-by-cases" tactic, we can simulate it by a lemma. If P is the formula being proved and x is a boolean subexpression of P, we use a lemma to rewrite P to $(x \Rightarrow P) \wedge (\neg x \Rightarrow P)$. In $x \Rightarrow P$, we can replace the subexpression x of P by any y that is known to be implied by x, and likewise for the other conjunct, that is reasoning by cases allows us to cross-fertilize P [5]. Using this lemma triggers additional rewriting activity to transform the left side to:

$$((\mathit{next}(A2) < A3) \Rightarrow (\neg\mathit{member}(A2, A1) \vee \neg(A1 < \mathit{next}(A2)))) \wedge \\ (\neg(\mathit{next}(A2) < A3) \Rightarrow (\neg\mathit{member}(A2, A1) \vee \neg(A1 < A3)))$$

The conjunct with antecedent $\mathit{next}(A2) < A3$ is rewritten as

$$(\mathit{next}(A2) < A3) \Rightarrow (\mathit{member}(A2, A1) \Rightarrow \neg(A1 < \mathit{next}(A2)))$$

fertilized with the inductive hypothesis, and reduced to *true*.

We continue by cases on $\mathit{member}(A2, A1)$ because the truth of the theorem depends on standard properties of ordering relations.

$$(\mathit{member}(A2, A1) \Rightarrow (\neg(\mathit{next}(A2) < A3) \Rightarrow ((A1 < \mathit{next}(A2)) \vee \neg(A1 < A3)))) \wedge \\ (\neg(\mathit{member}(A2, A1)) \Rightarrow (\neg(\mathit{next}(A2) < A3) \Rightarrow (\mathit{true} \vee \neg(A1 < A3))))$$

The first conjunct is rewritten to

$$member(A2, A1) \Rightarrow (\neg(next(A2) < A3) \wedge (\neg(A1 < next(A2)) \Rightarrow \neg(A1 < A3)))$$

where the consequent holds by the transitivity of "$\geq$". By successively reducing its consequents to *true*, the second conjunct is eventually reduced to *true*.

$$\begin{array}{l}\neg(member(A2, A1)) \Rightarrow (\neg(next(A2) < A3) \Rightarrow (true \vee \neg(A1 < A3)))\\ \neg(member(A2, A1)) \Rightarrow (\neg(next(A2) < A3) \Rightarrow true)\\ \neg(member(A2, A1)) \Rightarrow true\\ true\end{array}$$

Thus the proof terminates successfully.

The complexities of these proofs are comparable to those obtained with LP. All inductive variables are chosen automatically, less case analysis is required, and there is no need to invoke the Knuth-Bendix completion. Although this technique is occasionally useful, we find the proofs it generates difficult to understand, and thus we prefer this tactic for situations that do not allow data type induction, e.g., non constructor-based specifications.

We had to provide more lemmas to our prover. These lemmas are either trivial or instances of trivial lemma schemas, although suggesting them requires "understanding the proof." This is a mixed blessing. The effort to understand why a proof does or does not go through helps discovering the relevant properties of a specification. This may lead to better specifications and even code enhancements.

The user interface of our prover is very primitive. As a consequence we iterated our proof attempt several times before completion and we had to create manually the instances of the lemma schemas we supplied to the prover.

6 Concluding Remarks

We have discussed formal verification techniques for problems characterized by both differences in sizes and addressed properties. Loops are the critical components of small programs. The problems to be solved in this domain are correctness and termination, lack of expressiveness of the axiomatizations used for annotations, and the inherent difficulty of reasoning about repeated modifications of a program state.

Data type implementations are representative of medium size programs. The crucial problem to be solved is the mutual internal consistency of a group of related subroutines bound by the choice of the representation of abstract concepts by means of the structures offered by some hardware architecture and/or some programming language. Proofs of correctness in this domain entail not only the code, but also the representation mapping which has no physical presence in the software.

Module interconnection is the significant feature of large programs. Syntactic and semantic commitments of one component may not match the expectations of another. This problem is exacerbated by languages that allow the customization of a software unit by means of other units. Proving correctness does not involve code directly, but annotations generated by proofs of correctness for previous problems.

These tasks can all be addressed with a common formalism (equational specifications) and proof techniques (rewriting and induction). In particular, their formalizations are based on specifications that from a qualitative (and to some extent quantitative) point of view are independent of the tasks and of their sizes. Furthermore, we have discussed conceptual and practical tools for designing and using these specifications.

A crucial requirement of any specification is its adequacy. The assumption that a specification is "good" is often mistaken unless considerable care is devoted to its design. Several properties, unfortunately undecidable, are generally used to address the quality of a specification. We have shown that, by restricting the expressive power of our specification language, the most common and fundamental of these properties can be guaranteed. It is hard to say whether our restrictions are too severe, but it is encouraging to discover that typical verification problems proposed in the literature do not pose severe problems and that the proposed specifications are easy to use for both humans and an automated tool.

Rewriting is the fundamental idea behind our approach. The design strategies we have presented for designing rewrite rules ensure properties of the smaller units of specifications, the defined operations. Our strategies also allow us to build specifications incrementally in a way that preserves the properties of smaller units. In building large specifications from smaller ones, we glossed over the problems of modularizations and parameterization of specifications. Our approach is compatible with various techniques proposed for these features. In this respect, the properties we are able guarantee ease the composition of specifications.

The hardest task of nearly any verification problem is proving theorems. Informal proofs are easier to understand than formal ones, but are less reliable. Formal proofs, except for the simplest problems, are too complicated for humans without automated tools. Our proofs contain a few hundreds inferences, the majority of which are simple rewriting steps. Our prover becomes more effective with occasional hints. The lemmas we supply are "macro-steps" that the prover, for lack of knowledge and experience, would not carry to completion in certain contexts.

Finding the appropriate lemmas is not always easy. However, some lemmas such as those discussed in our comparison with LP are relatively standard. Others are suggested by the prover itself through generalization. From generalizations we sometimes find more elegant lemmas. Finally, by inspecting proof attempts, we are able to detect repeated patterns or formulas of increasing complexities which generally lead to proof failures. When these conditions arise, we look for lemmas that overcome the problems.

We believe that our specification approach is adequate for a large number of cases. However, our prover still fails to solve most non-trivial problems autonomously. It manages the bookkeeping of inductions and it provides hints for necessary lemmas. It completely removes the tedium of rewriting and the clerical mistakes associated with this activity. It prints readable proofs, although sometimes it makes inferences that are not necessary because the prover's rewrite strategy is innermost. An outermost rewriting strategy would produce shorter and more readable proofs. It shows a remarkable skill in finding inductive variables. Despite the considerable limitations in its user interface and proof tactics, the prover increases the quality of our specifications and enhances considerably the human ability to produce formal proofs for software problems.

References

[1] P. America. A parallel object-oriented language with inheritance and subtyping. *ACM SIGPLAN Notices*, 25(10):161–168, October 1990.

[2] S. Antoy. Automatically provable specifications. Technical Report 1876, Dept. of Computer Science, University of Maryland, 1987.

[3] S. Antoy. Design strategies for rewrite rules. In S. Kaplan and M. Okada, editors, *CTRS'90*, pages 333–341, Montreal, Canada, June 1990. Lect. Notes in Comp. Sci., Vol. 516.

[4] S.K. Basu. On development of iterative programs from functional specifications. *IEEE Trans. Soft. Eng.*, 6(2):170–182, 1980.

[5] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979.

[6] R. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[7] C. Choppy, S. Kaplan, and M. Soria. Complexity analysis of term-rewriting systems. *Theoretical Computer Science*, 67:261–282, 1989.

[8] J. de Bakker. *Mathematical Theory of Program Correctness.* Prentice-Hall, London, 1980.

[9] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[10] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North Holland, Amsterdam, 1990.

[11] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[12] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics.* Springer-Verlag, Berlin, 1985.

[13] S.J. Garland, J.V. Guttag, and J.J. Horning. Debugging Larch shared language specifications. *IEEE Trans. on Soft. Eng.*, 16(9):1044–1057, 1990.

[14] J. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, 1986.

[15] J. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational semantics of order-sorted algebras. In W. Brauer, editor, *CALP'85*, 1985. Lect. Notes in Comp. Sci., Vol. 194.

[16] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Menlo Park, CA, 1988.

[17] J. Guttag. Notes on type abstraction. *IEEE Trans. Soft. Eng.*, 6(1):13–23, 1980.

[18] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[19] J.V. Guttag, E. Horowitz, and D. Musser. Abstract data types and software validation. *Comm. of the ACM*, 21:1048–1064, 1978.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12:576–583, 1969.

[21] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[22] G. Huet. Confluent reductions: Abstract properties and applications to term-rewriting systems. *JACM*, 27:797–821, 1980.

[23] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. *JCSS*, 25:239–266, 1982.

[24] S. Kamin. The expressive theory of stacks. *Acta Informatica*, 24:695–709, 1987.

[25] D. Kapur, P. Narendran, and H. Zhang. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica*, 24:395–415, 1987.

[26] J.W. Klop. Term rewriting systems. In D. Gabbay S. Abramsky and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 1–112. Oxford University Press, 1992.

[27] D.E. Knuth and P.B. Bendix. *Simple word problems in universal algebras*, pages 263–297. Pergamon, 1970.

[28] G.T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[29] B. Liskov and J.M. Wing. Family values: A semantic notion of subtyping. Technical Report LCS TR-562, MIT, 1992.

[30] M. Wand. A new incompleteness result for Hoare's system. *JACM*, 25:168–175, 1978.

[31] W.A. Wulf, R.L. London, and M. Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Soft. Eng.*, 2:253–265, 1976.

[32] R.T. Yeh. Verification of programs by predicate transformation. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, volume 1, pages 228–247. Prentice-Hall, Englewood Cliff, NJ, 1978.

A Proving a Theorem

We show the session in which our prover proves the main theorem of example 2, that is, for all stacks s

$$concat(reverse(take(size(s), s)), newstack) = reverse(s)$$

The input to our prover has the syntax of Prolog. The complete input for the above problem is shown below. *Sortn* is a binary predicate declaring a sort and its constructors. *Function* is a ternary predicate defining arity and co-arity of a signature symbol. *Axiom* is a ternary predicate. It defines left and right sides of a rewrite rule in the second and third arguments respectively. The first argument is a label for reference in the proofs. *Theorem* is a predicate defining an equation to be proved. Similar to *axiom*, its first argument is a label. *Go* is the command to prove a set of theorems.

```
sortn(nat,[0,succ]).
function(0,[],nat).
function(succ,[nat],nat).

sortn(elem,[]).

sortn(stack,[newstack,push]).
function(newstack,[],stack).
function(push,[stack,elem],stack).

function(take,[nat,stack],stack).
axiom(take_1,take(0,_),newstack).
axiom(take_2,take(succ(_),newstack),newstack).
axiom(take_3,take(succ(N),push(S,E)),push(take(N,S),E)).

function(concat,[stack,stack],stack).
axiom(concat_1,concat(newstack,S),S).
axiom(concat_2,concat(push(S,E),T),push(concat(S,T),E)).

function(reverse,[stack],stack).
axiom(reverse_1,reverse(newstack),newstack).
axiom(reverse_2,reverse(push(S,E)),concat(reverse(S),push(newstack,E))).

function(size,[stack],nat).
axiom(size_1,size(newstack),0).
axiom(size_2,size(push(S,_)),succ(size(S))).

theorem('wp by pf', concat(reverse(take(size(S),S)),newstack), reverse(S)).

?- go.
```

The following transcript shows a session with our prover. *Appendixa* is the name of the file containing the above data. *Induce* is a shell script that loads a Prolog image of our prover and feeds input to it. Long lines have been wrapped to fit the page width. Lines headed by "IH-" show inductive hypotheses, if any. Lines headed by "(L)" or "(R)" indicate the application of a transformation to the left or respectively right side of the equation being proved. The transformation is explained in the string delimited by "<<".

The text describes how the lemma to which the proof reduces is automatically generated and how the proof could be considerably simplified by the use of two other lemmas.

```
antares[4]% induce appendixa

yes
{consulting /home/antares/users/antoy/theorems/appendixa...}

The theorem is:
  concat(reverse(take(size(A0),A0)),newstack) = reverse(A0)
Begin induction on A0
Induction on A0 case newstack
Inductive hypotheses are:
  (L) concat(reverse(take(size(newstack),newstack)),newstack)
         << subst A0 with newstack <<
  (R) reverse(newstack)     << subst A0 with newstack <<
  (L) concat(reverse(take(0,newstack)),newstack)     << reduct by size_1 <<
  (L) concat(reverse(newstack),newstack)     << reduct by take_1 <<
  (L) concat(newstack,newstack)     << reduct by reverse_1 <<
  (L) newstack     << reduct by concat_1 <<
  (R) newstack     << reduct by reverse_1 <<
  *** equality obtained ***
Induction on A0 case push(A1,A2)
Inductive hypotheses are:
  IH- concat(reverse(take(size(A1),A1)),newstack)=reverse(A1)
  (L) concat(reverse(take(size(push(A1,A2)),push(A1,A2))),newstack)
         << subst A0 with push(A1,A2) <<
  (R) reverse(push(A1,A2))     << subst A0 with push(A1,A2) <<
  (L) concat(reverse(take(succ(size(A1)),push(A1,A2))),newstack)
         << reduct by size_2 <<
  (L) concat(reverse(push(take(size(A1),A1),A2)),newstack)
         << reduct by take_3 <<
  (L) concat(concat(reverse(take(size(A1),A1)),push(newstack,A2)),newstack)
         << reduct by reverse_2 <<
  (R) concat(reverse(A1),push(newstack,A2))     << reduct by reverse_2 <<
  (R) concat(concat(reverse(take(size(A1),A1)),newstack),push(newstack,A2))
         << ind. hyp. on  A0 for A1 <<
  (L) concat(concat(A6,A7),newstack)
         << gen of reverse(take(size(A1),A1)) push(newstack,A2) <<
  (R) concat(concat(A6,newstack),A7)
         << gen of reverse(take(size(A1),A1)) push(newstack,A2) <<
  *** equality depends on next lemma ***
The lemma is:
  concat(concat(A6,A7),newstack) = concat(concat(A6,newstack),A7)
Begin induction on A6
Induction on A6 case newstack
Inductive hypotheses are:
  (L) concat(concat(newstack,A7),newstack)     << subst A6 with newstack <<
  (R) concat(concat(newstack,newstack),A7)     << subst A6 with newstack <<
  (L) concat(A7,newstack)     << reduct by concat_1 <<
  (R) concat(newstack,A7)     << reduct by concat_1 <<
  (R) A7     << reduct by concat_1 <<
Begin induction on A7
Induction on A7 case newstack
Inductive hypotheses are:
  (L) concat(newstack,newstack)     << subst A7 with newstack <<
  (R) newstack     << subst A7 with newstack <<
  (L) newstack     << reduct by concat_1 <<
  *** equality obtained ***
Induction on A7 case push(B0,B1)
Inductive hypotheses are:
  IH- concat(B0,newstack)=B0
  (L) concat(push(B0,B1),newstack)     << subst A7 with push(B0,B1) <<
  (R) push(B0,B1)     << subst A7 with push(B0,B1) <<
  (L) push(concat(B0,newstack),B1)     << reduct by concat_2 <<
  (L) push(B0,B1)     << ind. hyp. on  A7 for B0 <<
  *** equality obtained ***
End induction on A7
Induction on A6 case push(A8,A9)
Inductive hypotheses are:
  IH- concat(concat(A8,A7),newstack)=concat(concat(A8,newstack),A7)
  (L) concat(concat(push(A8,A9),A7),newstack)     << subst A6 with push(A8,A9) <<
  (R) concat(concat(push(A8,A9),newstack),A7)     << subst A6 with push(A8,A9) <<
  (L) concat(push(concat(A8,A7),A9),newstack)     << reduct by concat_2 <<
  (L) push(concat(concat(A8,A7),newstack),A9)     << reduct by concat_2 <<
  (R) concat(push(concat(A8,newstack),A9),A7)     << reduct by concat_2 <<
  (R) push(concat(concat(A8,newstack),A7),A9)     << reduct by concat_2 <<
  (L) push(concat(concat(A8,newstack),A7),A9)     << ind. hyp. on  A6 for A8 <<
  *** equality obtained ***
End induction on A6
End induction on A0
QED
{/home/antares/users/antoy/theorems/appendixa consulted, 1390 msec 4416 bytes}

yes
antares[5]%
```