Abstracting Pointers for a Verifying Compiler

Greg Kulczycki, Scott Duckworth, Murali Sitaraman, and Bruce W. Weide

Technical Report RSRG-06-01
Department of Computer Science
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

March 2006

Copyright © 2006 by the authors. All rights reserved.

Abstracting Pointers for a Verifying Compiler

Gregory Kulczycki[1], Scott Duckworth[2], Murali Sitaraman[2], and Bruce W. Weide[3]

[1] Computer Science, Virginia Tech, Falls Church, VA 22043, USA
gregwk@vt.edu

[2] Computer Science, Clemson University, Clemson, SC 29634, USA
(duckwos, murali}@cs.clemson.edu

[3] Computer Science and Engineering, Ohio State University, Columbus, OH 43210, USA
weide.1@osu.edu

Abstract. The ultimate objective of a verifying compiler is to prove that proposed code implements a full behavioral specification. Experience reveals this to be especially difficult for programs that involve pointers or references and linked data structures. In some situations, pointers are unavoidable; in some others, verification can be simplified through suitable abstractions. Regardless, a verifying compiler should be able to handle both cases, preferably using the same set of rules. To illustrate how this can be done, we examine two approaches to full verification. One replaces language-supplied indirection with software components whose specifications abstract pointers and pointer-manipulation operations. Another approach uses abstract specifications to encapsulate linked data structures that pointers and references are often used to implement, thereby limiting verification complications to inside the implementations of these components. Both approaches are applied to a sample program previously attacked using lightweight formal methods, focusing on problem-independent pointer properties, such as the absence of null references or cycles. The result is a basis for a compiler capable of full verification.

1 Introduction

There are at least two fundamental problems in developing a verifying compiler that is scalable. One of these is modularity. There is near universal agreement that to be scalable, the verification system must be modular, i.e., it must be possible for the verifying compiler to take just the *specifications* of components used by a piece of code and to establish that the purported implementation is correct with respect to its own specification. We take the requirement for modularity as given.

The other problem concerns the complexity of the assertions that are involved in the verification process. A variety of techniques have been explored in the literature to mitigate this. Most recent work involves "lightweight" formal methods, meaning a focus on specification-independent or easy-to-state *ad hoc* properties of an implementation. For example, such a system may be used to establish the absence of null dereferences [20] or

to demonstrate the absence of cycles in a pointer-based data structure [7]. Lightweight methods offer two benefits: they relieve programmers from the need to write full behavioral specifications and internal assertions such as loop invariants, and they show that progress toward the goal of full verification can be incremental.

While the lightweight approach is necessary and useful in the immediate term, verifiers ultimately need to become "heavyweight": to get beyond the point of showing merely that blatant error conditions do not exist, and to establish that programs actually achieve a full specification of desired behavior. A verifying compiler eventually must be able to deal with non-trivial assertions such as those needed to prove correctness of implementations that use pointers and references. For example, the sample code used to illustrate the lightweight methods of [7] is claimed to splice (interleave) two chains of nodes that look like linked lists; it should be possible both to state and to prove that it does. Several research efforts are focusing their attention on pointer verification problems like this, some by extending prior research in lightweight methods and others by focusing on more general verification [7][30].

This paper makes two contributions that fall in the latter category. First, we illustrate how previous results on modular reasoning about software components with full behavioral specifications can be extended to formal verification of programs that use indirection. We do this by showing how pointer-like behavior can be captured using a formally specified component interface with explicit operations to manipulate pointer-like variables. This approach allows the verifier to use the same set of proof rules [13][28] everywhere, whether or not pointers are involved. The complications associated with indirection arise only from the specifications of these particular components. Hence, they occur only where the required behavior of the program to be verified relies on the need for indirection, rather than permeating all proofs of correctness. The impact of this approach on verification is illustrated with a list-splicing example.

Regardless of the approach used for verification of pointer-like behavior, specifications and proofs turn out to be inherently complex compared to those not involving indirection [31]. So, in the second contribution of the paper, we illustrate that the list-splicing operation can be more easily specified and verified using an explicit list abstraction rather than directly manipulating the underlying pointer-based data structure. This approach simplifies not only specification and reasoning about the splicing code, but also the verification of other list operations, like the splice operation we use in our example, that otherwise would involve bare-handed manipulations of the underlying pointer-based data structures.

Our approach, being modular, includes a separate step—not part of this paper—showing that the list abstraction implemented using pointers is correct. While this proof is quite complex, the validity of the list abstraction as a cover story for some pointer manipulations needs to be established only once. Moreover, other correct implementations of the list abstraction can be substituted for the pointer-based one without the need for re-verifying the list-splicing code.

In short, we show why—with appropriate abstractions—it is not necessary to restrict the properties that can be proved by a verifying compiler about pointer-based data structures to simple or *ad hoc* ones. The paper is organized as follows. Section 2 gives an overview of the verifying compiler we envision. Section 3 introduces a pointer abstraction. It gives a specification for a list-splicing operation that takes pointer-like variables (rather than language-supplied pointers) as parameters, and shows how to verify the implementation. Section 4 explains the list abstraction and discusses reasoning about an

implementation of the Spice operation based on that abstraction. It contains a sample proof of an obligation using the Coq prover [6]. Section 5 discusses related work and summarizes the paper.

2 Overview of the Verifying Compiler

We are currently building a verifying compiler with an overall architecture illustrated in Figure 1. The system takes as its inputs the specification of a component or operation and an implementation. The input includes the context for verification, such as the mathematical units that contain both the definitions used in the assertions and the theorems necessary to complete proofs of correctness. To facilitate modular verification, the context should also include specifications (but not implementations) of components that are reused in the implementation. For mechanizing the verification process, the system demands that implementations are annotated with suitable assertions, such as abstraction relations and representation invariants for data representations, and loop invariants and progress metrics for loops. The system is sound and it checks that programmer-supplied assertions, such as invariants, hold before employing them in proofs.

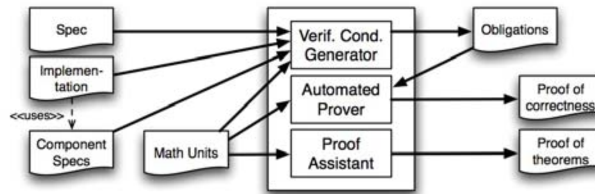

Fig. 1. A portion of the architecture for the verifying compiler

Beyond a sophisticated type checker for mathematical assertions, specifications, and implementations, the verification system includes a verification condition generator for establishing the correctness of a given implementation with respect to its specification, an automated prover that discharges proofs automatically, and a proof assistant that checks steps of a manually-developed proof. The automated prover will be used to prove the correctness of (simpler) verification conditions automatically, while the proof assistant is one in the spirit of Coq that is more suited for proving non-trivial mathematical theorems for use by the automated prover. Our conjecture here is that given a sufficiently rich collection of mathematical definitions and theorems, the correctness proofs would require little more than the basic rules of logic and substitution, possibly with some "hints" provided by program annotations.

3 Approach for Reasoning about a Pointer-Based Implementation

This section illustrates how the verifying compiler employs its rules for reasoning about the correctness of a pointer-based implementation of a Splice operation [7]. Informally, the Splice operation takes as input two pointer variables, p and q, that each begin a distinct, singly-linked list of locations. The length of q's list must be less than or equal to the

length of *p*'s list. The operation modifies the first list so that it is a perfect shuffle of the locations in the original lists. A shuffled list contains all the elements of both lists with their original orderings preserved, similar to what happens when you shuffle a deck of cards. A perfect shuffle interleaves elements from each list. The effect of the procedure call *Splice*(*p*, *q*) can be seen informally in Figure 2.

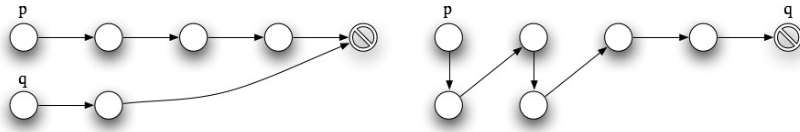

Fig. 2. The effect of the call Splice(p, q) on a system of linked locations

The formal specification of the Splice operation given in this section uses a pointer component. The verification system assumes that all programming types, variables, and operations are defined in a component. This applies not only to user-defined types, but also to types that are considered "built-in" to the language itself, such as booleans, integers, arrays, and even pointers. The advantage of this is that reasoning about all types is uniform. In particular, reasoning about pointers involves the same value-based reasoning system used for other types—no special proof rules or semantics are required. Pointers and other built-in types, however, may have special syntax and implementations. For example, programmers can reason about the pointer assignment of *p* to *q* as a call to *Relocate*(*p*, *q*), but the compiler will implement it as a simple machine instruction that overwrites a memory address.

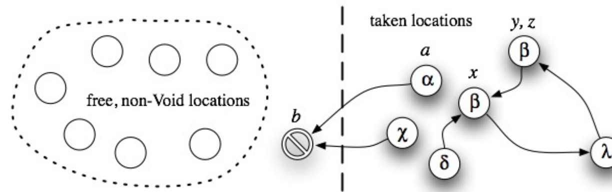

Fig. 3. A system of linked locations in which locations contain symbols (Greek letters) and links that point to other locations. Locations are either free or taken (allocated), and a special *Void* location is perpetually free. Position variables (pointers) are denoted by Roman letters.

A formal specification of an interface to capture pointer behavior is discussed in [17], and a portion of its specification is given in the appendix. Informally, the pointer component describes a system of linked locations that contain information and links to other locations, as illustrated in Figure 3. To capture the behavior of a system of linked locations, the concept defines and uses three global, conceptual variables (similar to [4][19]): *Contents*, *Target*, and *Is_Taken*. *Contents*(*q*) is the information at location *q*, *Target*(*q*) is the location targeted by *q*'s link, and *Is_Taken*(*q*) is true if and only if *q* is allocated, and therefore, taken. Conceptual variables are not programming variables, but they are an

essential part of the program state, which is defined by our semantics as just the abstract values of the defined programming and conceptual variables at any point in the program.

```
Definition Var Is_Reachable_in(hops: N; p, q: Location): B =
   Target^hops(p) = q and ∀k: N, if Target^k(p) = q then k ≥ hops;
Definition Var Is_Reachable(p, q: Location): B = ∃k: N ∋
Is_Reachable_in(k, p, q);
Definition Var Distance(p, q: Location): N  =
 { k   if Is_Reachable_in(k, p, q)
 { 0   otherwise                      ;
Definition Var Is_Info_Str(p, q: Location; α: Str(Info)): B =
   ∃n: N ∋ Is_Reachable_in(n, p, q) and α = Π_{k=1}^{n} ⟨Contents(Target^k (p))⟩;
Operation Splice(preserves p: Position; clears q: Position);
   updates Target;
   requires ( ∃k1, k2: N ∋ Is_Reachable_in(k1, p, Void) and
                          Is_Reachable_in(k2, q, Void) and k2 ≤ k1 )
and
      ( ∀r: Location, if (Is_Reachable(p, r) and Is_Reachable(q, r))
then r = Void );
   ensures ( ∀t: Location, if not Is_Reachable(#p, t) and not
Is_Reachable(#q, t)
                            then Target(t) = #Target(t) ) and
      ( ∀α, β, γ: Str(Info), if Is_Info_Str(p, Void, α) and
                            Is_Info_Str(#p, Void, β) and
                            Is_Info_Str(#q, Void, γ) then α ≤!≥ (β, γ)
);
Recursive procedure
   decreasing Distance(q, Void);
   Var r, s: Position;
   If (not At_Void(q)) then
      Relocate(r, p);
      Follow_Link(r);
      Relocate(s, q);
      Follow_Link(s);
      Redirect_Link(p, q);
      Redirect_Link(q, r);
      Splice(r, s);
      Clear(q);
   end;
end;
```

Fig. 4. Pointer-based specification and implementation of the Splice operation.

3.1 Pointer-Based Specification

Figure 4 gives a specification and an implementation for Splice. The specification defines several mathematical functions used in other parts of the specification. *Is_Reachable_in*(n, p, q) is true if and only if location q is reachable from location p in n hops. If x is a variable at location p, this function is true if and only if x will arrive at location q for the first time by following its link exactly n times. The term $Target^k(p)$ in the definition denotes k applications of the function *Target* to p. *Is_Reachable*(p, q) is true iff q is reachable from p in any number of hops. When the *Var* keyword follows *Definition*, it indicates that the value of the function may vary for the same input values in different program states. The *Distance* between p and q is the number of hops it takes to get from p to q provided that q is reachable from p; otherwise, the distance is zero.

The Splice operation preserves p and clears q. In other words, p is unchanged and q is Void after the operation. The *updates* clause indicates that Target is the only global variable that may be modified by an invocation of the Splice operation. The verification system adheres to an implicit frame property [3] that restricts the effects of an operation invocation to the actual parameters and any globals listed in the *updates* clause. Since conceptual variables are considered to be part of the state space, they must be included in the updates clause if an invocation can modify their value.

The *requires* clause is fulfilled only if the linked lists beginning at p and q are acyclic and disjoint. The only way that a location can reach Void is if there are no cycles in the linked structure beginning with that location. Provided that lists are free of cycles, k_1 and k_2 represent the lengths of lists p and q, respectively, and k_1 must be greater than or equal to k_2. Finally, if any location other than Void is reachable by both p and q, then the lists are not disjoint.

The *ensures* clause has two main conjuncts. The first conjunct indicates which portions of the *Target* variable do not change. It asserts that the links of locations do not change for locations that are not part of either input list. We know that the contents and the taken status of all locations in the system are not affected by this operation because the variables *Contents* and *Is_Taken* are not included in the *updates* clause. The second main conjunct in the *ensures* clause describes how the lists are modified. Essentially it says that α is the string of Info objects derived from the output list, β and γ are the strings of Info objects derived from the two input lists, and α is a perfect shuffle (or "interleaving") of β and γ, which we denote by $\alpha \leq ! \geq (\beta, \gamma)$. The recursive procedure includes a *decreasing* metric that allows the verifier to confirm that the recursive call structure eventually terminates. Note that a #v in the ensures clause always indicates the incoming value of the variable v.

3.2 Pointer-Based Reasoning

A symbolic reasoning table [9][28] for the Splice procedure is given in Table 1. For each state, the table shows a path condition, facts, and obligations. The path condition must hold for the program to enter the specified state, the facts tell us what we know about the values of the variables in that state, and the obligations tell us what needs to be true before we can move to the next state.

Table 1. A partial reasoning table for establishing correctness of the Splice procedure

St	Path cond	Facts	Obligations
0		r_0 = Void **and** s_0 = Void **and** ($\exists k_1, k_2$: **N** ∋ Is_Reachable_in(k_1, p_0, Void) **and** Is_Reachable_in(k_2, q_0, Void) **and** $k_2 \leq k_1$) **and** ($\forall$t: Location, **if** Is_Reachable(p_0, t) **and** Is_Reachable(q_0, t) **then** r = Void)	
If (**not** At_Void(q)) **then** --			
1	$q_0 \neq$ Void	$r_1 = r_0$ **and** $s_1 = s_0$ **and** …	
Relocate(r, p); --			
2	$q_0 \neq$ Void	$r_2 = p_2$ **and** $s_2 = s_1$ **and** …	Is_Taken$_2$(r_2)
Follow_Link(r); --			
3	$q_0 \neq$ Void	r_3 = Target$_3$(r_2) **and** $s_3 = s_2$ **and** …	
Relocate(s, q); ---			
4	$q_0 \neq$ Void	$r_4 = r_3$ **and** $s_4 = q_4$ **and** …	Is_Taken$_4$(s_4)
Follow_Link(s); --			
5	$q_0 \neq$ Void	$r_5 = r_4$ **and** s_5 = Target$_5$(s_4) **and** …	Is_Taken$_5$(p_5)
Redirect_Link(p, q); ---			
6	$q_0 \neq$ Void	Target$_6$ = Target$_5${p_6 a q_6} **and** $r_6 = r_5$ **and** $s_6 = s_5$ **and** …	Is_Taken$_6$(q_6)
Redirect_Link(q, r); --			
7	$q_0 \neq$ Void	Target$_7$ = Target$_6${q_7 a r_7} **and** $r_7 = r_6$ **and** $s_7 = s_6$ **and** …	pre$_{Splice}$[p ~> r_7, q ~> s_7] **and** Distance(s_7, Void) < Distance(q_0, Void)
Splice(r, s); --			
8	$q_0 \neq$ Void	post$_{Splice}$[#p ~> r_7, p ~> r_8, #q ~> s_7, q ~> s_8] **and** …	
Clear(q); --			
9	$q_0 \neq$ Void	$r_7 = r_6$ **and** $s_7 = s_6$ **and** $p_7 = p_6$ **and** q_7 = Void **and** …	

end; ---			
10a	q_0 = Void	$r_8 = r_0$ **and** $s_8 = s_0$ **and** $p_8 = p_0$ and $q_8 = q_0$ **and** $\text{Target}_8 = \text{Target}_0$ **and** $\text{Contents}_8 = \text{Contents}_0$ **and** $\text{Is_Taken}_8 = \text{Is_Taken}_0$	(∀t: Location, **if** **not** Is_Reachable(p_0, t) **and** **not** Is_Reachable(q_0, t) **then** Target(t) = #Target(t)) **and** (∀α, β, γ: Str(Info), **if** Is_Info_Str(p_8, Void, α) **and** Is_Info_Str(p_0, Void, β) **and** Is_Info_Str(q_0, Void, γ) **then** α ≤!≥ (β, γ))
10b	$q_0 \neq$ Void	$r_8 = r_7$ **and** $s_8 = s_7$ **and** …	-- same as 10a --

For example, the only way the program can reach state 5 is if *q* does not have a value of *Void* in state 0, so the path condition for state 5 is "$q_0 \neq$ Void". State 5 occurs just after the call *Follow_Link*(*s*). The postcondition of *Follow_Link* (see Figure A-1) indicates that the value of *s* in state 5 will be equal to the value of the Target function applied to the value of *s* in state 4, or "$s_5 = \text{Target}_5(s_4)$". Since *Follow_Link*(*s*) only has *s* as a parameter and the declaration of *Follow_Link* does not list any globals in its updates clause, we know that no other variables have been modified, so that $r_5 = r_4$ and $p_5 = p_4$, and so on. The statement just after state 5 is *Redirect_Link*(*p*, *q*). The requires clause for this operation indicates that *p* must be at a taken location before the client can call it. Therefore, we have an obligation to show that "Is_Taken(p_5)" before we can move to the next state.

In general, obligations come from preconditions of called operations and facts come from postconditions of called operations. An important exception to this are the facts in state 0, which come from the precondition of the Splice operation itself, and the obligations in the last state (10a and 10b), which come from the postcondition of the Splice operation. State 7 includes an obligation that "Distance(s_7, Void) < Distance(q_0, Void)", which comes from the *decreasing* clause of the Splice procedure, and allows us to prove that the recursion terminates. The two final states in this table correspond to the two ways that the program could have reached this state: either by going through the *then* block of the *if* statement ($q_0 \neq$ Void) or by skipping it altogether (q_0 = Void).

We abbreviated a few items to make the table to fit on one page. For example, we normally list all current variables in the facts column, even if they are unchanged. In this table, we use ellipses to indicate that unlisted variables have not changed. The obligation in state 7 comes from the precondition of the Splice operation. We abbreviated this by writing "$\text{pre}_{\text{Splice}}[p \sim> r_7, q \sim> s_7]$", meaning the precondition of Splice where *p* is replaced by r_7 and *q* is replaced by s_7. We do something similar in state 8 with the postcondition of Splice. Finally, we use the notation "$\text{Target}_5\{p_6$ a $q_6\}$" to mean that the variable Target is unchanged except that location p_6 maps to q_6.

The symbolic reasoning table shown here can be generated automatically by the verification system. To prove that the implementation of Splice is correct with respect to its specification, one must discharge all of the obligations in the table. We give an example of proving obligations in the next section.

4 Abstracting Pointer-Based Data Structures

Though the pointer-based implementation of the Splice operation is necessarily intricate, reasoning about its correctness does not involve any special axioms, proof rules, or semantics for pointers. Every part of the specification and reasoning adheres to a simple value-based semantics in which (a) the state space is described exactly by the currently defined variables—programming or conceptual—and their values, and (b) the portion of the state space affected by an operation invocation is restricted to its actual parameters and the globals listed in its updates clause. The conceptual variables add complexity to the specification and reasoning, but this complexity is a necessary part of the pointer component, and reasoning about them is no different than reasoning about global *programming* variables.

Despite the uniformity in specification and reasoning for pointers, there is an aspect of the pointer-based Splice operation that is *unnecessarily* complex, namely, the fact that it is implemented using pointers in the first place. The Splice operation as specified above is essentially an operation on singly-linked lists. We can therefore simplify the reasoning about the Splice operation by making it an operation on list objects rather than pointer objects.

Unfortunately, it is not sufficient to hide the standard pointer-based implementation of a list using an interface to achieve "pointer-free" reasoning. For example, a Java list class hides its implementation, but programmers cannot avoid reasoning about pointers (in Java's case, references) because variables denote references rather than values and both insertions and deletions involve references of objects. Therefore, to capture the Java List component behavior properly, it is necessary to model a List (and an Object) as a mathematical location and define a global, conceptual "heap" variable that is potentially updated in every operation (see [16]). The result is specification and reasoning of complexity not unlike Fig. 3, compromising the very abstraction we were trying to achieve. Modeling and language support for avoiding references and aliasing are discussed in [8][16].

```
Operation Splice(updates P: List; clears Q: List);
    requires |Q.Prec| = 0 and |P.Rem| ≥ |Q.Rem|;
    ensures ∃α: Str(Entry) ∋ α ≤!≥ (#P.Rem,
#Q.Rem) and
            P.Prec = #P.Prec ○ a and |P.Rem| = 0;
Recursive procedure Splice(updates P: List;
clears Q: List);
    decreasing |Q.Rem|;
    Var E: Entry;
    If (Rem_Length(Q) /= 0) then
        Advance(P);
        Remove(E, Q);
        Insert(E, P);
        Advance(P);
        Splice(P, Q);
    end;
end;
```

Fig. 5. List-based specification and implementation of the Splice operation

4.1 List-Based Specification and Reasoning

Figure 5 shows a list-based Splice operation and its implementation. Here, we use the conceptualization of lists given in the appendix (and explained in [28]), where a List is modeled as an ordered pair: a preceding and a remaining string of entries. The front of the remaining string serves as the imaginary list insertion point. Using this modeling, the specification states that on a call to *Splice*(*P*, *Q*) with List $P = \langle a, b\rangle\langle x, y, z\rangle$ and List $Q = \langle\rangle\langle\alpha, \beta\rangle$, the result will be $P = \langle a, b, x, \alpha, y, \beta, z\rangle\langle\rangle$ and $Q = \langle\rangle\langle\rangle$.

Table 2. A complete reasoning table for list-based Splice procedure

St	Path cond	Facts	Obligations
0		Entry.**is_initial**(E_0) **and** $\lvert Q_0.Prec\rvert = 0$ **and** $\lvert P_0.Rem\rvert \geq \lvert Q_0.Rem\rvert$	
If (Rem_Length(Q) /= 0) **then** ----------			
1	$\lvert Q_0.Rem\rvert \neq 0$	$E_1 = E_0$ **and** $P_1 = P_0$ **and** $Q_1 = Q_0$	$\lvert P_1.Rem\rvert > 0$
Advance(P); ----------			
2	$\lvert Q_0.Rem\rvert \neq 0$	$E_2 = E_1$ **and** $Q_2 = Q_1$ **and** $P_2.Prec \circ P_2.Rem =$ $P_1.Prec \circ P_1.Rem$ **and** $\lvert P_2.Prec\rvert = \lvert P_1.Prec\rvert + 1$	$\lvert Q_2.Rem\rvert > 0$
Remove(E, Q); ----------			
3	$\lvert Q_0.Rem\rvert \neq 0$	$P_3 = P_2$ **and** $Q_3.Prec = Q_2.Prec$ **and** $Q_2.Rem = \langle E_3\rangle \circ Q_3.Rem$	
Insert(E, P); ----------			
4	$\lvert Q_0.Rem\rvert \neq 0$	$P_4.Prec = P_3.Prec$ **and** $P_4.Rem = \langle E_3\rangle \circ P_3.Rem$ **and** $Q_4 = Q_3$ **and** $E_4 = ??$	$\lvert P_4.Rem\rvert > 0$
Advance(P); ----------			
5	$\lvert Q_0.Rem\rvert \neq 0$	$E_5 = E_4$ **and** $Q_5 = Q_4$ **and** $P_5.Prec \circ P_5.Rem =$ $P_4.Prec \circ P_4.Rem$ **and** $\lvert P_5.Prec\rvert = \lvert P_4.Prec\rvert + 1$	$\lvert Q_5.Prec\rvert = 0$ **and** $\lvert P_5.Rem\rvert \geq \lvert Q_5.Rem\rvert$ **and** $\lvert Q_5.Rem\rvert < \lvert Q_0.Rem\rvert$
Splice(P, Q); ----------			
6	$\lvert Q_0.Rem\rvert \neq 0$	$E_6 = E_5$ **and** $\lvert Q_6.Prec\rvert = 0$ **and** $\lvert Q_6.Rem\rvert = 0$ **and** $\exists\alpha$: Str(Entry) $\ni$ $\alpha \leq!\geq (P_5.Rem, Q_5.Rem)$ **and** $P_6.Prec = P_5.Prec \circ \alpha$ **and** $\lvert P_6.Rem\rvert = 0$	
end; ----------			
7a	$\lvert Q_0.Rem\rvert = 0$	$E_7 = E_0$ **and** $P_7 = P_0$ **and** $Q_7 = Q_0$	$\lvert Q_7.Prec\rvert = 0$ **and** $\lvert Q_7.Rem\rvert = 0$ **and** $\exists\alpha$: Str(Entry) $\ni$ $\alpha \leq!\geq (P_0.Rem, Q_0.Rem)$ **and** $P_7.Prec = P_0.Prec \circ \alpha$ **and** $\lvert P_7.Rem\rvert = 0$
7b	$\lvert Q_0.Rem\rvert \neq 0$	$E_7 = E_6$ **and** $P_7 = P_6$ **and** $Q_7 = Q_6$	-- same as 7a --

The decreasing clause allows the verifier to establish termination by proving that the list decreases in length with each recursive call. The symbolic reasoning table for the list-based Splice implementation is given in Table 2. It is much simpler than the symbolic reasoning table in Table 2, and therefore we do not abbreviate here.

4.2 Proof of Correctness

For illustration, we present here a mechanical, step-by-step walk through of the proof of the obligation in state 1 using the Coq proof assistant. It is also possible to do the proof using other provers, such as PVS [24] with which we have experimented or ACL2 [12] which we are exploring. The obligation we consider is somewhat non-trivial because we need to establish that it is appropriate to advance list P, though the *if* condition only guarantees that Q can be advanced. The obligation is converted into the theorem below to be established by Coq prover. In specifying the theorem, to be consistent with the notations used in this paper, we have enhanced Coq with a minimal string library to define the type `string`. Given this, we now have to prove the following theorem.

```
Coq < Theorem splice_obligation_1:
  forall Q0 P0 P1 : string Entry * string Entry,
   |P0 Rem| >= |Q0 Rem| -> |Q0 Rem| <> 0 -> P1 = P0 ->
      |P1 Rem| > 0.
```

Shown below are proof steps for completing a formal proof using Coq.

```
splice_obligation_1 < intros.
splice_obligation_1 < replace P1 with P0.
splice_obligation_1 < apply (neq_O_ge_gt (|Q0 Rem|)(|P0
Rem|)).
splice_obligation_1 < assumption.
splice_obligation_1 < assumption.
splice_obligation_1 < Qed.
```

In the proof, the first instruction "intros" essentially makes "`| P1 Rem | > 0`" the goal to be deduced from a sequence of hypotheses through repeated application of the deduction theorem. Following the replacement, we apply a locally-defined theorem, the proof of which has been omitted.

```
Coq < Theorem neq_O_ge_gt : forall n m : nat,
  n <> 0 -> m >= n -> m > 0.
```

Application of the theorem leads to subgoals and these are among the hypotheses assumed by Coq immediately after intros. The two subsequent assumption instructions allow Coq to use the hypotheses and complete the proof.

Now consider the obligation in state 7. The formal proof is more intricate, so we merely give a proof sketch here for the case when $|Q_0.Rem| \neq 0$. All variables in state 7 are unchanged from state 6, so conjuncts $|Q_7.Prec| = 0$, $|Q_7.Rem| = 0$, and $|P_7.Rem| = 0$ all

follow directly from the facts in state 6. Therefore, it suffices to show that $P_7.Prec = P_0.Prec \circ \alpha_7$ where $\alpha_7 \leq ! \geq (P_0.Rem, Q_0.Rem)$. To facilitate this, let $P_0.Rem = \langle x \rangle \circ \sigma$ and $Q_0.Rem = \langle y \rangle \circ \tau$, where x and y are entries and σ and τ are strings of entries. We know this is possible since $|P_0.Rem| \geq |Q_0.Rem| > 0$. Thus, it suffices to show that

$$P_7.Prec = P_0.Prec \circ \alpha_7 \textbf{ where } \alpha_7 \leq ! \geq (\langle x \rangle \circ \sigma, \langle y \rangle \circ \tau)$$

From state 6, we know $P_6.Prec = P_5.Prec \circ \alpha_6$ where $\alpha_6 \leq ! \geq (P_5.Rem, Q_5.Rem)$. With a bit of effort we can show that $P_5.Prec = P_0.Prec \circ \langle x \rangle \circ \langle y \rangle$, $P_5.Rem = \sigma$, and $Q_5.Rem = \tau$. Once we do, the assertion from state 6 becomes $P_6.Prec = P_0.Prec \circ \langle x \rangle \circ \langle y \rangle \circ \alpha_6$ where $\alpha_6 \leq ! \geq (\sigma, \tau)$. Substituting the value of $P_6.Prec$ for $P_7.Prec$ in the equation above and eliminating the string $P_0.Prec$ from both sides yields

$$\langle x \rangle \circ \langle y \rangle \circ \alpha_6 = \alpha_7 \textbf{ where } \alpha_6 \leq ! \geq (\sigma, \tau) \textbf{ and } \alpha_7 \leq ! \geq (\langle x \rangle \circ \sigma, \langle y \rangle \circ \tau)$$

This can be proven as a theorem in the math unit describing string theory, but one can see that this statement is true based on our informal explanation of perfect shuffle.

5 Discussion

The Splice example is taken from a recent paper on shape analysis by Hackett and Rugina [7] in which they use a region-based shape analysis algorithm to show that the C code for Splice does not introduce cycles into lists. We use the pointer component to formalize this lightweight property in [17] and discuss the differences between lightweight and heavyweight specifications for the operation in the context of a loop-based implementation. Other tools can prove selected properties involving pointers and references. The ESC/Java tool [20] is used to statically detect heap-related errors in Java. The Alloy approach [30] examines heap-based properties in an effort to discover faulty implementations of linked data structures.

Practitioners and researchers have successfully encapsulated certain aspects of pointer behavior in generic C++ classes. Safe pointers [21] and checked pointers [25] focus on eliminating errors that arise from the complexities of manual memory management. In contrast, the focus of the pointer component used here is on uniformity in specification and reasoning, particularly in the context of a verifying compiler based on value-semantics. Suitable abstractions like the list component allow programmers to avoid reasoning about complex memory management issues just as it allows them to avoid reasoning about indirection. The inclusion of operations that allow manual memory management in the pointer component used here facilitate our research into incorporating performance reasoning into the verification system [14][27].

Many object-oriented languages avoid most memory errors by using automatic garbage collection. However, in many cases these languages have exacerbated the reasoning problem caused by aliasing [11], which can break encapsulation and thwart modular reasoning [5]. As illustrated here, models that capture pointer behavior complicate formal specification and verification [31]. Therefore, various proposals have been introduced to control object aliasing [5][11][22]. Reasoning about programs that incorporate these techniques is typically done in the context of object-oriented logics that use a global heap abstraction [1][23], even when specification languages supply more sophisticated abstractions [2][18]. The LOOP compiler for Java/JML [29] translates programming code and specifi-

cations into a heap-based logic before discharging proof obligations. In our value-based approach, formal reasoning occurs at the same level of abstraction provided by the specification. Maintaining sophisticated abstractions throughout the verification process helps automatic provers and human understanding alike.

A complete formal specification of the pointer component described here can be found in [15], and a detailed discussion of the list component can be found in [28]. Future research includes exploring how to develop lightweight and heavyweight performance specifications [14][27] for the pointer component and linked data structures, and reason about these specifications in the context of a verifying compiler.

Acknowledgements

We would like to acknowledge Bill Ogden for his insights into the design of the pointer specification. This work is funded in part by the U. S. National Science Foundation grant CCR-0113181 and a grant from the U. S. National Aeronautics and Space Administration through the SC Space Grant Consortium.

References

1. Abadi, M., Leino, K. R. M.: A Logic of Object-Oriented Programs. In: Bidoit, M., Dauchet, M. (eds.): TAPSOFT ’97: Theory and Practice of Software Development, 7th International Joint Conference. Springer-Verlag, New York (1997) 682–696
2. Barnett, M., Leino, K. R. M., Schulte, W.: The Spec# Programming System: An Overview. In: CASSIS 2004, LNCS Vol. 3362. Springer (2004)
3. Borgida, A., Mylopoulos, J. Reiter, R.: … And Nothing Else Changes?: The Frame Problem in Procedure Specifications. In: Proceedings of the 15th International Conference on Software Engineering. IEEE Computer Society Press (1993) 303–314
4. Cheon, Y., Leavens, G. T., Sitaraman, M., Edwards, S.: Model Variables: Cleanly Supporting Abstraction in Design by Contract. Software, Practice, and Experience, Vol. 35, No. 6. (2005) 583–599
5. Clarke, D. G., Potter, J. M., Noble, J.: Ownership Types for Flexible Alias Protection. In: Proceedings OOPSLA ’98. (1998) 48–64
6. Dowek, G., Felty, A., Herbelin, H., Huet, G.: The COQ Proof Assistant User’s Guide. Technical Report 154, Inria-Rocquencourt, France (1993)
7. Hackett, B. Rugina, R.: Region-Based Shape Analysis with Tracked Locations. In: Proceedings POPL ’05. (2005)
8. Harms, D. E., Weide, B. W.: Copying and Swapping: Influences on the Design of Reusable Software Components. IEEE Transactions on Software Engineering, Vol. 17, No. 5 (1991) 424–435
9. Heym, W.: Computer Program Verification: Improvements for Human Reasoning. Ph.D. thesis, The Ohio State University (1995)
10. Hoare, C. A. R., Misra, J.: Verified Software: Theories, Tools and Experiments. In: IFIP Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE). Zurich (2005) http://vstte.ethz.ch/.

11. Hogg, J., Lea, D., Wills, A., deChampeaux, D., Holt, R.: The Geneva Convention on the treatment of object aliasing. OOPS Messenger, Vol. 3, No. 2. (1992) 11–16
12. Kaufmann, M., Manolios, P., Moore, J. S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
13. Krone, J.: The Role of Verification in Software Reusability. Ph.D. thesis, The Ohio State University (1988)
14. Krone, J., Ogden, W. F., Sitaraman, M.: Modular Verification of Performance Correctness. In: OOPSLA 2001 SAVCBS Workshop Proceedings. (2001) http://www.cs.iastate.edu/ ~leavens/SAVCBS/papers-2001/index.html.
15. Kulczycki, G., Sitaraman, M., Ogden, W. F., Hollingsworth, J. E.: Component Technology for Pointers: Why and How. Technical Report RSRG-03-03, Clemson University. (2003) http://www.cs.clemson.edu/~resolve/reports/RSRG-03-03.pdf
16. Kulczycki, G., Sitaraman, M., Ogden, W. F., Weide, B. W.: Clean Semantics for Calls with Repeated Arguments. Technical Report RSRG-05-01, Clemson University (2005) http://www.cs.clemson.edu/~resolve/reports/RSRG-05-01.pdf
17. Kulczycki, G., Sitaraman, M., Weide, B. W., Rountev, A.: A Specification-Based Approach to Reasoning about Pointers. In: Proceedings ESEC/FSE SAVCBS '05 Workshop. ACM Software Engineering Notes, Vol. 31, No. 2 (2005)
18. Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., Cok. D. R.: How the Design of JML Accommodates both Runtime Assertion Checking and Formal Verification. Science of Computer Programming, Vol. 55. Elsevier (2005) 185–205
19. Leino, K. R. M.: Data Groups: Specifying the Modification of Extended State. In: Proceedings OOPSLA '98. (1998) 144–153
20. Leino, K. R. M., Nelson, G., Saxe, J. B.: ESC/Java User's Manual. Technical Note 2000-002, Compaq Systems Research Center (2000)
21. Meyers, S.: More Effective C++. Addison-Wesley (1995)
22. Müller, P., Poetzsch-Heffter, A.: Modular Specification and Verification Techniques for Object-Oriented Software Components. In: Leavens, G. T., Sitaraman, M. (eds.), Foundations of Component-Based Systems. Cambridge University Press, Cambridge, UK (2000)
23. O'Hearn, P., Reynolds, J., Yang, H.: Local Reasoning about Programs that Alter Data Structures. Lecture Notes in Computer Science, Vol. 2142 (2001) 1–19
24. Owre, S., Rushby, J. M., Shankar, N.: PVS: A Prototype Verification System. In: Kapur, D. (ed.), 11th Intl. Conf. on Automated Deduction. LNCS, Vol. 607 (1992) 748–752
25. Pike, S. M., Weide, B. W., Hollingsworth, J. E.: Checkmate: Concerning C++ Dynamic Memory Errors with Checked Pointers. In: Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education. ACM Press (2000)
26. Sitaraman, M., Weide, B.W.: Component-Based Software using RESOLVE. ACM Software Engineering Notes, Vol. 19, No. 4. (1994) 21–67
27. Sitaraman, M.: Impact of Performance Considerations on Formal Specification Design. Formal Aspects of Computing, Vol. 8, No. 6. (1996) 716–736
28. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., Hollingsworth, J. E.: Reasoning about Software-Component Behavior. In: Proceedings of the 6th International Conf. on Software Reuse. Springer-Verlag (2000) 266–283

29. van den Berg, J., Jacobs, B.: The LOOP Compiler for Java and JML. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, Springer (2001)
30. Vaziri, M., Jackson, D.: Checking Heap-Manipulating Procedures with a Constraint Solver. TACAS ’03. Warsaw, Poland (2003)
31. Weide, B.W., Heym, W.D.: Specification and Verification with References. In: Proc. OOPSLA Workshop on Specification and Verification of Component-Based Systems. (2001)

Appendix

```
Concept Location_Linking_Template (type Info);
    Defines Location: Set;
    Defines Void: Location;
    Var Target: Location → Location;
    Var Contents: Location → Info;
    Var Is_Taken: Location → B;

    Initialization ensures ∀q: Location, ¬Is Taken(q);
    Constraints ¬Is Taken(Void) and (∀q: Location,
        if ¬Is Taken(q) then Info.Is_Initial(Contents(q)) and
        Target(q) = Void) and …
    Type Family Position is modeled by Location;
        exemplar p;
        Initialization ensures p = Void;

    Operation Take_New_Location(updates p: Position);
        …
    Operation Abandon_Location(clears p: Position);
        …
    Operation Relocate(updates p: Position; preserves q: Position);
        ensures p = q;

    Operation Follow_Link(updates p: Position);
        requires Is Taken(p);
        ensures p = Target(#p, i);

    Operation Redirect_Link(preserves p: Position; preserves q:
Position);
        updates Target;
        requires Is_Taken(p);
        ensures ∀r: Location, Target(r) = { q            if r = p
                                          { # Target(r)  otherwise   ;

    Operation At_Void(preserves p: Position): Boolean;
        …
end Location_Linking_Template;
```

Fig. A-1. A portion of a specification for a generic pointer component. It defines a set Location that includes a distinguished Void element, and it declares three global, conceptual variables that contribute to the state space of the program. This is a simplification of a more generalized component that is also parameterized by the number of outgoing links from each location. The *updates* parameter mode indicates that the operation modifies this argument; the *clears* mode ensures that the argument will return from the procedure with an initial value of its type; and the *preserves* mode prohibits any changes to the argument's value. All operations and their specifications are given in [15]. The specification language used is RESOLVE [26][28].

Concept List_Template (**type** Entry);

    Type Family List **is modeled by** (Prec: Str(Entry) × Rem: Str(Entry));
        exemplar S;
        Initialization ensures |S.Prec| = 0 and |S.Rem| = 0;

    Operation Insert(**alters** E: Entry; **updates** S: List);
        ensures S.Prec = #S.Prec **and** S.Rem = ⟨#E⟩ ○ #S.Rem;

    Operation Remove(**replaces** R: Entry; **updates** L: List);
        requires |S.Rem| > 0;
        ensures S.Prec = #S.Prec **and** #S.Rem = ⟨R⟩ ○ S.Rem;

    Operation Advance(**updates** S: List);
        requires |S.Rem| > 0;
        ensures S.Prec ○ S.Rem = #S.Prec ○ #S.Rem and
            |S.Prec| = |S.Prec| + 1;

    Operation Reset(**updates** S: List);
        ensures |S.Prec| = 0 **and** S.Rem = #S.Prec ○ #S.Rem;

    -- other list operations …

end List_Template;

Fig. A-2. A portion of a specification for a list component that models the list as two mathematical strings. A more complete list specification and a discussion of how to reason with it is given in [28].