A Specification-Based Approach to Reasoning About Pointers

Gregory Kulczycki
Virginia Tech
Falls Church, VA
gregwk@vt. edu

Murali Sitaraman
Clemson University
Clemson, SC
murali@cs.clemson.edu

Bruce W. Weide
Atanas Rountev
The Ohio State University
Columbus, OH
weide@cse.ohio-state.edu
rountev@cse.ohio-state.edu

ABSTRACT

This paper explains how a uniform, specification-based approach to reasoning about component-based programs can be used to reason about programs that manipulate pointers. No special axioms, language semantics, global heap model, or proof rules for pointers are necessary. We show how this is possible by capturing pointers and operations that manipulate them in the specification of a software component. The proposed approach is mechanizable as long as programmers are able to understand mathematical specifications and write assertions, such as loop invariants. While some of the previous efforts in reasoning do not require such mathematical sophistication on the part of programmers, they are limited in the kinds of properties they can prove about programs that use pointers. We illustrate the idea using a "Splice" operation for linked lists, which has been used previously to explain other analysis techniques. Not only can the proposed approach be used to establish shape properties given lightweight specifications, but also it can be used to establish total correctness given more complete specifications.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *correctness proofs, formal methods.* D.3.3 [**Programming Languages**]: Language Constructs and Features – *data types and structures.*

General Terms

Languages, Verification.

Keywords

Pointer specification, reasoning, heap memory management.

1. INTRODUCTION

Reasoning about program code involving pointers or references is notoriously difficult [29]. Various logics have been developed for object-oriented languages such as Java and C# in which references are implicit [1][18]. Reasoning about programs in these languages is complicated due to the possibility of aliasing. Various static analysis techniques also have been applied to languages with and without explicit deallocation (e.g., [6][22][30]). These techniques are fast and flexible, but they are also limited in what they can prove about a program's run-time behavior. Both object-oriented logics and general program analysis techniques tend to rely on global reasoning about entire heap abstractions. Frame properties [3] make reasoning about heap locations somewhat less demanding in object-oriented logics, and a separation logic [20] has been suggested as a way to further localize reasoning to portions of the heap structure. Nonetheless, a single heap abstraction is still assumed. Building on previous work in shape analysis [30], Hackett and Rugina [6] describe an approach that avoids global heap abstractions. Instead, it uses local reasoning about individual heap locations to find potential errors.

In this paper, we consider from a language design perspective the problem of pointers and reasoning about programs that use them. We describe a way to implement and reason about programs involving pointers by using a formally specified generic component that encapsulates pointer-like behavior and that is especially well suited for the implementation of linked data structures. Furthermore, no global heap abstraction is used in reasoning about pointers. Instead, shared conceptual (or specification-only) variables whose scope is at the component level are defined in the specification. These variables record the state of the pointer structure, keeping track of such information as which locations are mapped to which objects and how the locations are linked to one another. The component permits explicit deallocation and thereby allows users to reason about memory errors that do not arise with garbage collection, so it can accommodate situations and languages where no automatic garbage collection is assumed. Section 2 briefly describes the specification of the pointer component abstraction and its operations.

A key contribution of the specification-based approach to reasoning about programs with pointers is that programmers can use and reason about pointers using the same techniques that they use to reason about all other components in a program. This does not preclude a language designer from inventing special syntax for pointers as long as the meaning of that syntax can be described in terms of the operations specified in the component. In particular, for the component to exhibit the run-time performance of language-supplied pointers, a compiler for a language with component-provided pointers need not implement these component operations as typical calls, but may consider them to be built-in constructs. For example, even though a programmer using our component will reason about the pointer assignment statement "*p* = *q*" as a call *Relocate(p, q)*—a procedure call that must conform to the contract specified by its precondition and postcondition—the compiler may implement this statement as a single machine instruction that overwrites *p*'s value with the address stored in *q*.

A second contribution of the specification-based approach is that it facilitates more powerful reasoning about properties of pointer-based programs than previous static analyses relying on special rules to handle language-supplied pointers. This is the topic of Section 3. In that section, we illustrate the issues using an exam-

ple "Splice" operation for a linked list. The example is taken from a recent paper on a general approach to shape analysis [6]. In that paper, Hackett and Rugina introduce and use a region-based shape analysis algorithm to establish the "shape property" that the Splice code does not introduce cycles into lists. They describe a non-trivial algorithm that partitions memory into regions, keeping track of the relationships between regions using a unification-based points-to analysis [27] that they augment with context sensitivity. Individual "configurations" are used to track the state of individual heap locations. These configurations can be analyzed independently of each other, eliminating the need to keep track of how an entire heap abstraction changes over the course of a program's execution.

Shape analysis is fully automatic and, unlike our specification-based approach, does not require programmer-supplied assertions. However, the authors note, for example, that their analysis would not apply if the Splice code were written slightly differently. More importantly, shape analysis techniques—and other static analysis techniques—are limited in the kinds of properties they can be used to prove. We illustrate these issues using both lightweight and heavyweight specifications for the "Splice" operation. Whereas the lightweight specification is sufficient to prove the assertion that an implementation of Splice does not introduce cycles, a more complete specification of the operation shows the potential of the pointer specification approach to analyze the full behavior of the operation and its implementation.

2. SPECIFYING POINTER BEHAVIOR

A formal specification of a component to capture pointer behavior is given in the technical report [12], where the design rationale for the specification and performance ramifications are discussed. The specification is general and it allows reasoning about any pointer-based data structures, including lists and trees. A skeleton of this specification is discussed in this subsection as a prelude to specification-based reasoning about pointers.

2.1 Mathematical Modeling

Without loss of generality, the specification in Figure 1 is given in the RESOLVE notation [23][25]. The specification *defines* Location as a mathematical set. The exact set of addresses that correspond to locations is an internal implementation detail, and it is suppressed in the specification. For the purposes of reasoning, the client programmer need only know that Location is a set and Void is a specific location element from that set. At any given program state, some locations are *free*, or available for allocation; and some locations are *taken*, or already allocated. A key aspect of the specification is to formalize how locations become linked to each other following various pointer manipulation operations. Hence the name *Location_Linking_Template* for the concept.

The concept is parameterized by the type of information associated with each location and the number of links from each location. If the nodes of a list contain GIF pictures, for example, then Info is a type representing GIF pictures. Similarly, the number of links depends on the application. For example, a singly linked list requires one link from each location, whereas a *k*-ary tree requires *k* links from each location.

To capture the behavior of a system of linked locations, the concept defines and uses three global, conceptual variables: *Contents*(*q*) is the information at a given location *q*, *Target*(*q*, *i*) is the location targeted by the *i*-th link of *q*, and *Is_Taken*(*q*) is true if and only if a given location *q* is allocated, and therefore, taken. These variables are not programming variables; they are used solely for specification and reasoning. They are similar to specification-only variables used in other formalisms for object-oriented programs [4][14].

```
Concept Location_Linking_Template (type Info;
                                   evaluates k: Integer);
    Defines Location: Set;
    Defines Void: Location;

    Var Target: Location × [1..k] → 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
        ∀j: [1..k], Target(q, j) = 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;
                          evaluates i: Integer);
        requires Is Taken(p) and 1 ≤ i ≤ k;
        ensures p = Target(#p, i);

    Operation Redirect_Link(preserves p: Position;
                  evaluates i: Integer; preserves q: Position);
        updates Target;
        requires 1 ≤ i ≤ k and Is_Taken(p);
        ensures ∀r: Location, ∀j: [1..k],
                Target(r, j) = { q              if r = p and j = i
                               { # Target(r, j) otherwise          ;

    Operation Check_Colocation(preserves p, q: Position;
                               replaces are_colocated: Boolean);
                                   …
    Operation Swap_Locations(preserves p: Position;
            evaluates i: Integer; updates new_target: Position);
        …
    Operation Swap_Contents(preserves p: Position;
                            updates I: Info);
        …
    Operation Is_At_Void(preserves p: Position): Boolean;
        …
    Operation Location_Size(): Integer;
        …
end Location_Linking_Template;
```

Figure 1. A Skeleton of Pointer Behavior Specification.

The concept is constrained to behave as specified in the invariant constraints clause: the Void location can never be taken or allocated, i.e., it is always free; and all locations that are freely available are in an initialized state, i.e., their contents have default information and their links point to Void. Finally, the number of locations available is related to the total available memory capacity. Hence, locations are limited, and allocations without corresponding deallocations will eventually deplete the pool. Initially, all locations are assumed to be free and, therefore, initialized, as specified in the **constraints** clause. This does not mean that an implementation of the pointer component must initialize every location at the beginning; of course, it would not. It just means that any newly allocated location is guaranteed to be initialized, and this objective can be achieved as and when it is needed.

Given this model, a programming variable of type Position is simply viewed mathematically as a location. When a programmer declares a new pointer variable, it is initially just the Void location. Only after allocation does the variable become a location that can contain information.

A system of linked locations is established when a client instantiates the pointer component with the type of information that each location holds and the number of links coming from each location. Figure 2 gives an informal view of an example system of linked locations where type Info is assumed to be Greek letters and the number of links from each location is assumed to be one. Here, the circles represent locations that contain information (Greek letters) and a fixed number of links (just one in the example) to other locations.

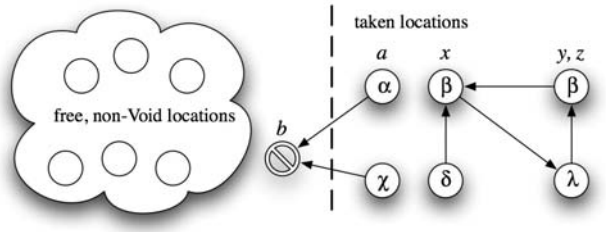

Figure 2. A system of linked locations.

The figure shows that some of the locations are taken and others are free. All the pointer variables (*a*, *b*, *x*, *y*, *z*) except for *b* are at allocated locations. For example, the information content at the location of variable *x* is β. The pointer variables *y* and *z* are colocated, i.e., they are aliased, and the link from that location points to the location of *x*. The pointer variable *b* resides at the special Void location, which is perpetually free. Due to poor programming or possible reliance on a garbage collector, some of the allocated locations have become inaccessible, such as the locations containing information χ and δ. To manipulate pointer variables to reach a state like the one in Figure 2, a programmer has to declare pointer variables and call suitable operations, as explained in the next subsection.

2.2 Discussion of Pointer Operations

The parameters in the specifications of operations in Figure 1 use various modes to help the programmer understand rough effects of a call to the arguments before reading the subsequent formal specification. The **updates** 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; the **preserves** mode prohibits any changes to the argument's value; the **replaces** mode indicates that the incoming argument value will be ignored but replaced; and the **evaluates** mode indicates that the operation expects an expression in this position—it is typically used with types that are often returned from functions, such as integers.

The *Take_New_Location* operation allows a programmer to associate information with a specified pointer variable. Every call to this operation leads to a new location being taken. Internally, this operation allocates memory for a new object of type Info and makes *p* point to it. A taken location remains taken until the client abandons it, which she can do using the *Abandon_Location* operation. When a location is abandoned, memory for the information it contained is reclaimed and the pointer variable that is being explicitly abandoned is repositioned to the Void location. The *Location_Size* operation helps a programmer determine if there is sufficient memory for a new allocation, i.e., if there are any more free locations available for taking. A careful programmer may need this operation to check availability before calling the *Take_New_Location* operation.

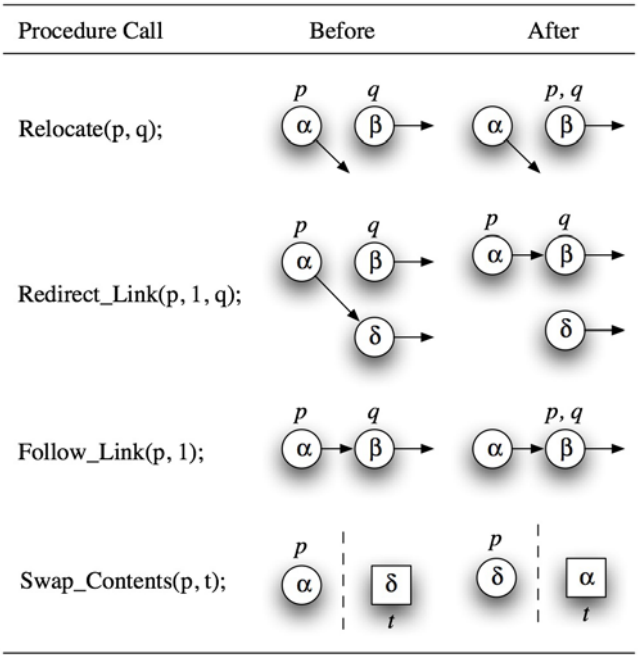

Figure 3. The effects of selected calls on a system.

Now we consider the formal specification of operation *Redirect_Link*. This operation redirects the *i*-th link at position *p*'s location to *q*'s location. The values of both *p* and *q* are preserved, since both occupy the same locations they did before the operation is invoked. The **updates** clause after the formal parameters lists any component-level conceptual variables that are affected by the operation. In this case, the *Target* variable will be modified, but the *Contents* and the *Is_Taken* variables will remain unchanged. The RESOLVE specification language adheres to an implicit frame property in that operation invocations may only affect explicit parameters to the operation or component conceptual variables listed in the **updates** clause. In this case, the operation specifies that the values of both position parameters are preserved and that any integer parameter is treated as an expression,

so the only variable that is modified is *Target*. The postcondition describes how the *Target* variable is modified: the *Target* function does not change except that it now maps the tuple (*p*, *i*) to *q*. Note that the hash (#) symbol denotes the incoming value of a variable. The specifications for *Relocate* and *Follow_Link* are straightforward.

The remaining operations allow a client to manipulate links and information at occupied locations. They also allow position variables to be associated with different locations. Figure 3 gives before and after views of a system for invocations of selected operations. Note that the box in the *Swap_Contents* operation is not a location. It simply indicates the value of Info variable *t*. In this case, *t*'s value is α before the operation and δ after the operation. In contrast, *p*'s value (which is a location) remains the same before and after the operation, but the *Target* variable will have been updated. All the operations shown here have formal specifications associated with them [12]. We have shown only a few of these in Figure 1, owing to space constraints.

Because the pointer component supports explicit deallocation, memory errors can arise within the context of a system. A location is considered accessible if there is a path to that location from a location occupied by some position variable. In Figure 2, for example, two taken locations are not accessible: the location containing δ and the location containing χ. The location containing λ is accessible even though it is not occupied by a position variable because there is a path to it from the location occupied by *x*. A location that is taken but not accessible is a memory leak; a location (other than Void) that is free but accessible is a dangling reference. The latter situation is not, perforce, a memory error; but it becomes one if that pointer variable is then used before being updated.

3. EXAMPLE

This section illustrates how the pointer component can be used for both lightweight and heavyweight specifications and subsequent reasoning. The Splice operation takes as input two singly-linked lists of locations: one that begins with a location occupied by position *p* and another that begins with a location occupied by position *q*. 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. If location *x* appears before location *y* in one of the original lists, then *x* appears before *y* in the shuffled list. A perfect shuffle interleaves elements from each of the lists.

3.1 Simple Splice Specification

Figure 5 gives a lightweight specification and code for Splice (a minor syntactic variation of the version in [6]). The specification is sufficient to meet the goal of a typical shape analysis for the operation, namely to "statically verify that, if the input lists […] are disjoint and acyclic, then the [output list] is acyclic" [6] (page 3). Note that we use a syntactic shortcut throughout this section for ease of reading (and writing) the specifications. Since all locations in these examples have exactly one link, we leave out the link number where it is typically required. For example, we use *Target*(*p*) instead of *Target*(*p*, 1).

The specification defines two mathematical functions used in the specification. *Is_Reachable_in*(*n*, *p*, *q*) is true if and only if location *q* is reachable from location *p* in *n* hops. That is, if *x* is a variable at location *p*, this function is true if and only if *x* will arrive at location *q* by following his first link *n* times, but no fewer. The term $Target^k(p)$ means *k* iterations of the function *Target* starting with *p*. The requirement that "**if** $Target^k(p) = q$ **then** k ≥ hops" for all k, guarantees, for example, that *Is_Reachable_in*(10, *p*, *q*) is false if *q* links back to itself and it only takes 5 hops for *p* to reach *q*. *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. For example, *Is_Reachable*(*p*, *q*) may be true in one state and false in the next if one of the links between them was redirected. The *Distance* between *p* and *q* is the number of hops it takes to get from *p* to *q* if *q* is reachable from *p*; otherwise, the distance is zero.

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**

$$= \begin{cases} k & \text{if Is\_Reachable\_in(k, p, q)} \\ 0 & \text{otherwise} \end{cases};$$

Operation Splice(**preserves** p: Position; **clears** q: Position);
  updates Target;
  requires ($\exists k_1, k_2$: **N** ϶ Is_Reachable_in(k_1, p, Void) **and**
    Is_Reachable_in(k_2, q, Void) **and** $k_2 \le k_1$) **and**
    (∀r: Location, **if** Is_Reachable(p, r) **and**
      Is_Reachable(q, r) **then** r = Void);
  ensures Is_Reachable(p, Void);
Procedure
  Var r: Position;
  Var s: Position;
  Relocate(r, p);
  While (**not** At_Void(q))
    decreasing Distance(q, Void);
    maintaining Is_Reachable(p, Void);
  do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
  end;
end Splice;

Figure 5. A lightweight specification for Splice.

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 conceptual variable that is modified.

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. The Void location always links to itself. 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 simple **ensures** clause is true when the output list p is acyclic.

Figure 6 illustrates the execution of the splice operation when p is a linked list with four locations and q is a linked list with two locations. Part (a) represents the state of the system at the beginning of the first loop, part (b) represents the system at the beginning of the second loop iteration, and part (c) represents the state of the system when the loop terminates.

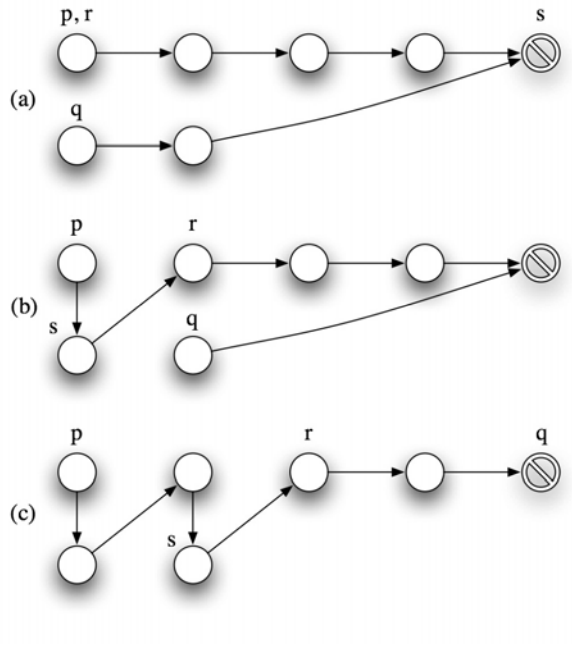

Figure 6. An animation of the implementation for Splice.

The correctness of the implementation can be proved using the formal proof system detailed in [7][10] and summarized in [25]. The proof process takes the programmer-supplied invariant for the loop, establishes that it is invariant, and employs it in completing the proof. For the present example, the loop invariant asserts that it is always possible to reach the Void location from p. It is obviously true at the beginning of the first iteration, since p does not change and we know from the precondition that Void is reachable from p. All that is left to prove is that the invariant holds from one iteration to the next. This follows from the fact that the following lemmas hold in each state in the loop.

Lemma #1: Is_Reachable(q, Void);
Lemma #2: Is_Reachable(r, Void);
Lemma #3: Is_Reachable(p, r) **or** Is_Reachable(p, q);

The proof of correctness of Splice follows from the invariant and the negation of the loop condition. For proving termination, the process uses a progress metric given in the **decreasing** clause. The progress metric states that the "distance" from q to Void decreases with each iteration of the loop. This argument establishes the same result as the intricate shape analysis proposed in [6]. A limitation of the Splice operation as specified in this section is that it cannot be used as a meaningful guide to anyone who implements the operation. In fact, even an implementation that does nothing at all will guarantee the postcondition because it follows directly from the precondition. The next section provides a more detailed specification for the Splice operation.

3.2 Full Splice Specification

The full specification of the Splice operation is given in Figure 5.

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**

$$= \begin{cases} k & \text{if Is\_Reachable\_in}(k, p, q) \\ 0 & \text{otherwise} \end{cases};$$

Definition Var Is_Info_Str(p, q: Location; α: Str(Info)): **B** =
∃n: N ϶ Is_Reachable_in(n, p, q) **and**
$\alpha = \prod_{k=1}^{n} \langle \text{Contents}(\text{Target}^{k}(p)) \rangle$;

Operation Splice(**preserves** p: Position; **clears** q: Position);
updates Target;
requires (∃k_1, k_2: **N** ϶ Is_Reachable_in(k_1, p, Void) **and**
Is_Reachable_in(k_2, q, Void) **and** $k_2 \le k_1$) **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));

Procedure
Var r: Position;
Var s: Position;
Relocate(r, p);
While (**not** At_Void(q))
decreasing Distance(q, Void);
maintaining (∀t: Location, **if not** Is_Reachable(#p, t) **and**
not Is_Reachable (#q, t) **then** Target(t) = #Target(t)) **and**
(∀χ, δ, ε. β, γ, ρ: Str(Info), **if** Is_Info_Str(p, r, χ) **and**
Is_Info_Str(r, Void, δ) **and** Is_Info_Str(q, Void, ε) **and**
Is_Info_Str(#p, Void, β) **and** Is_Info_Str(#q, Void, γ) **and**
ρ ≤!≥ (δ, ε) **then** χ o ρ ≤!≥ (β, γ));
do
Relocate(s, r);
Follow_Link(r);
Redirect_Link(s, q);
Follow_Link(s);
Follow_Link(q);
Redirect_Link(s, r);
end;
end Splice;

Figure 7. A full specification for Splice.

We assume the definitions given above and introduce another one: *Is_Info_Str*. The function *Is_Info_Str*(p, q, α) is true if and only if q is reachable from p and α is the string of all the objects of type Info contained in the locations between p and q. The

string includes the object in *p* but not the object in *q*. For example, for the system of linked locations in Figure 2, *Is_Info_Str*(*x*, *y*, ⟨β, λ⟩) is true.

The **requires** clause in this specification has not changed from the last one. However, the **ensures** clause is more detailed. It 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. Note that 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 α ≤!≥ (β, γ). Recall that a perfect shuffle of two strings is a shuffle that interleaves the first *n* elements of each string, where *n* is the length of the shorter string. A perfect shuffle always starts with the first element of the first string. For example, ⟨a, e, b, f, c, d⟩ is a perfect shuffle of the strings ⟨a, b, c, d⟩ and ⟨e, f⟩.

Since the postcondition has been strengthened, the loop invariant also needs to be stronger. Like the **ensures** clause, the loop invariant is divided into two main conjuncts. The first conjunct simply mirrors the first part of the **ensures** clause. The second conjunct asserts that if ρ is a perfect shuffle of the linked lists beginning with *r* and *q*, then when the string beginning with *p* and ending with *r* is concatenated with ρ, the resulting list is a perfect shuffle of the input lists. For convenience, we have chosen strings β and γ to designate the same strings in the invariant as they do in the **ensures** clause. The invariant includes a few extra strings: χ, which is the string of Info objects between *q* and Void; δ, which is the string of objects between *r* and Void; and ε, which is the string of objects between *q* and Void.

At the beginning of the first iteration of the loop, χ is the empty string, while δ = β and ε = γ. So, when ρ ≤!≥ (δ, ε), we also know that χ o ρ ≤!≥ (β, γ). When the loop terminates, *q* = Void, so ε represents the empty string and therefore ρ = δ and χ o ρ = χ o δ. But χ o δ is the same as α in the ensures clause, so that α ≤!≥ (β, γ) follows directly from χ o ρ ≤!≥ (β, γ).

Finally, the proof of correctness for the Splice operation must show that χ o ρ ≤!≥ (β, γ) is maintained in the invariant. If we assume that χ o ρ ≤!≥ (β, γ) at the beginning of some arbitrary iteration, we must then show that χ′ o ρ′ ≤!≥ (β, γ) at the beginning of the next iteration, where χ′ and ρ′ are the new values of χ and ρ. (Note that β and γ are based on #*p* and #*q*, so they do not change from one iteration to the next.) Since *r* and *q* both advance exactly one location, we know that δ = ⟨*x*⟩ o δ′ and ε = ⟨*y*⟩ o ε′ for some Info objects *x* and *y*. A perfect shuffle of δ′ and ε′ will be the same as a perfect shuffle of δ and ε except that it will no longer hold *x* and *y*. In other words, ρ′ = ⟨*x*⟩ o ⟨*y*⟩ o ρ. While ρ loses these objects, the Info string χ picks them up. In the code, position *s* moves to the location containing *x*, redirects the link there to the location containing *y*, follows the link, and then redirects the link at that location toward *r*'s new location. As a result of this traversal, χ′ = χ o ⟨*x*⟩ o ⟨*y*⟩. When χ′ and ρ′ are concatenated we get χ′ o ρ′ = χ o ρ, so that χ o ρ ≤!≥ (β, γ) implies χ′ o ρ′ ≤!≥ (β, γ).

Of course, a formal proof of the invariant would be much more intricate, but this should give the reader an idea of how to proceed.

4. DISCUSSION

Using programmer-supplied loop invariants (similar to our approach for handling loops), Jenson et al. have discussed in [9] how to prove heap-related properties and find counterexamples to claimed properties. Their implementation has been shown to be effective in practice. Their work differs from traditional pointer analyses because they can answer more questions that can be expressed as properties in first-order logic. While this work focuses on linear linked lists and tree structures, more recently Møller and Schwartzbach have extended the results to all data structures that can be expressed as "graph types" [17]. The Alloy approach "targets properties of the heap" [28] in a quest to root out erroneous implementations of linked data structures and null dereferences. The ESC/Java tool [15] has the ability to statically detect heap-related errors in Java. Though we have focused only on Hackett and Rugina's work in this paper, there is significant other work in shape analysis, including work on parametric shape analysis that allows more questions to be answered concerning heaps [22]. None of these efforts is based on a general, formal specification of pointer behavior.

The idea of capturing pointer behavior in the form of a component is not new. Safe pointers [16] and checked pointers [21] are generic C++ classes designed to alleviate memory errors in C++ by implementing all or part of the memory management code inside a pointer-like data structure. In contrast, our pointer specification supports manual memory management and the memory errors that can occur as the result of it. Though we have focused only on proving properties and correctness through reasoning, the results can be combined with previous work [26] to identify errors through analysis. In addition, these errors are statically predictable in the context of a formal specification and verification system that does not treat reasoning about pointers different from reasoning about any other component.

Despite the fact that many object-oriented languages avoid most memory errors by using automatic garbage collection, implicit pointers (references) remain a serious problem for both formalists and practitioners. This is due primarily to aliasing [8]. Aliasing—in the absence of a complete model of pointers and their referents—breaks encapsulation [19] and hence thwarts modular reasoning [5]. When pointers are appropriately modeled, formal specification and verification is complicated because the model must cope with soundness [29]. Therefore, various proposals have been introduced to control object aliasing, such as [5][19]. Reasoning about programs that incorporate these techniques is typically done in the context of object-oriented logics that use a global heap abstraction.

A complete formal specification of the pointer component described here was omitted due to space considerations, but it can be found in [12]. Future research includes exploring how the specification can be adapted for languages with automatic garbage collection, and how we can develop both lightweight and

heavyweight performance specifications [11][24] towards analyzing the performance of pointer-based programs.

5. ACKNOWLEDGMENTS

We would like to acknowledge Bill Ogden for his insights into the design of the pointer specification. We would also like to thank Kunal Chopra and Jason Mashburn for intricate discussions of the topics in this paper. This work is funded in part by the National Science Foundation grant CCR-0113181.

6. REFERENCES

[1] Abadi, M. and Leino, K. R. M. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th Internation Joint Conference*, pages 682–696. Springer-Verlag, New York, 1997.

[2] Barnett, M., Leino, K. R. M., and Schulte, W. The Spec# programming system: An overview. In *CASSIS 2004*, LNCS vol. 3362, Springer, 2004.

[3] Borgida, A., Mylopoulos, J., and 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, pages 303–314, 1993.

[4] Cheon, Y., Leavens, G. T., Sitaraman, M., and Edwards, S. Model variables: Cleanly supporting abstraction in design by contract. *Software, Practice, and Experience*, 35 (6), pages 583–599, 2005.

[5] Clarke, D. G., Potter, J. M., and Noble, J. Ownership types for flexible alias protection. In *Proceedings OOPSLA '98*, pages 48–64, 1998.

[6] Hackett, B. and Rugina, R. Region-based shape analysis with tracked locations. In *Proceedings POPL '05*, January 2005.

[7] Heym, W. *Computer Program Verification: Improvements for Human Reasoning*. Ph.D. thesis, The Ohio State University, 1995.

[8] Hogg, J., Lea, D., Wills, A., deChampeaux, D., and Holt, R. *The Geneva Convention on the treatment of object aliasing*. OOPS Messenger, 3(2):11–16, 1992.

[9] Jensen, J. L., Jorgensen, M. E., Klarlund, N., and Schwartzbach, M. I. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings SIGPLAN Conference on Programming Language Design and Implementation, 1997*.

[10] Krone, J. *The Role of Verification in Software Reusability*. Ph.D. thesis, The Ohio State University, 1988.

[11] Krone, J., Ogden, W. F., and 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.

[12] Kulczycki, G., Sitaraman, M., Ogden, W. F., and Hollingsworth, J. E. *Component Technology for Pointers: Why and How*, Technical Report RSRG-03-03, Clemson University, Clemson, SC. 2003. http://www.cs.clemson.edu/~resolve/reports/RSRG-03-03.pdf

[13] Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., and Cok. D. R. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, vol. 55, pages 185-205, Elsevier, 2005.

[14] Leino, K. R. M. Data groups: specifying the modification of extended state. In *Proceedings OOPSLA '98*, pages 144–153, 1998.

[15] Leino, K. R. M., Nelson, G., and Saxe, J. B. *ESC/Java User's Manual*, Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[16] Meyers, S. *More Effective C++*. Addison-Wesley, 1995.

[17] Møller, A. and Schwartzbach, M. I. The pointer assertion logic engine. In *ACM SIGPLAN Notices*, 36(5), pages 221–231, May 2001.

[18] Müller, P. and Poetzsch-Heffter, A. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, editors, Cambridge University Press, Cambridge, United Kingdom, 2000.

[19] Noble, J., Vitek, J., and Potter, J. Flexible alias protection. *ECOOP '98*. Lecture Notes in Computer Science, vol. 1445, pp. 158–185, 1998.

[20] O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, 2142:1–19, 2001.

[21] Pike, S. M., Weide, B. W., and 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, March 2000.

[22] Sagiv, M., Reps, T., and Wilhelm, R. Parametric shape analysis via 3-valued logic. *In ACM Transactions on Programming Languages and Systems*, 24(3), pp. 217–298, 2002.

[23] Sitaraman, M. and Weide, B.W. Component-based software using RESOLVE. *ACM Software Engineering Notes*, 19(4), pp. 21–67, 1994.

[24] Sitaraman, M. Impact of performance considerations on formal specification design. *Formal Aspects of Computing*, 8(6):716–736, 1996.

[25] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W. Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E. Reasoning about software-component behavior. In *Proceedings of the 6th International Conference on Software Reuse*, pages 266–283. Springer-Verlag, 2000.

[26] Sitaraman, M., Gandi, D. P., Kuechlin, W., Sinz, C., and Weide, B. W. DEET for Component-Based Software, *In Proceedings FSE Workshop on Specification and Verification of Component-Based Systems*, October 2004.

[27] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, January 1996.

[28] Vaziri, M. and Jackson, D. Checking heap-manipulating procedures with a constraint solver. *TACAS '03*, Warsaw, Poland, 2003.

[29] Weide, B.W., and Heym, W.D. Specification and verification with references. In *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems*, October 2001.

[30] Wilhelm, R., Sagiv, M., and Reps, T. Shape analysis. *In Proceedings of the 2000 International Conference on Compiler Construction*, Berlin, Germany, April 2000.

[31] Wing, J. M. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.