Automatic Assume/Guarantee Reasoning for Heap-Manipulating Programs (Ongoing Work)

Greta Yorsh [a] Alexey Skidanov [a] Thomas Reps [b] Mooly Sagiv [a]

[a] *School of Comp. Sci., Tel-Aviv Univ., {gretay,skidanov,msagiv}@post.tau.ac.il*

[b] *Comp.Sci.Dept., Univ. of Wisconsin., reps@cs.wisc.edu*

Abstract

Assume/Guarantee (A/G) reasoning for heap-manipulating programs is challenging because the heap can be mutated in an arbitrary way by procedure calls. Moreover, specifying the potential side-effects of a procedure is non-trivial. We report on an on-going effort to reduce the burden of A/G reasoning for heap-manipulating programs by automatically generating post-conditions and estimating side-effects of non-recursive procedures. Our method is sound. It combines the use of theorem provers and abstract-interpretation algorithms.

Key words: assume-guarantee reasoning, side-effect, mod-clauses, shape analysis, abstract interpretation, theorem prover

1 Introduction

Shape-analysis algorithms statically analyze a program to determine information about the heap-allocated data structures that the program manipulates. The algorithms are *conservative* (sound), i.e., the discovered information is true for every input. The analysis of large programs presents a major problem for existing shape analyzers. This ongoing work investigates one possibility for scaling shape analysis to handle larger programs using *assume/guarantee reasoning*. The main idea is to require the programmer to specify (some aspects of) the behavior of every procedure (also called a contract), and to apply shape analysis to analyze every procedure in isolation, using the contracts of other procedures. However, contracts impose a burden on the programmer. In particular, specifying the potential side-effects of a procedure is a complex issue because of the need to express the pre- and post-conditions of a method independent of the calling context [5].

This research investigates an orthogonal alternative, which does not assume that the potential side-effects are specified by the programmer. Instead, we only require that a pre-condition of every method be specified, and that the code not include recursive calls.[1] This allows us to apply abstract-interpretation algorithms [2] to compute for every procedure: (a) a conservative over-approximation of the post-condition of the method, and (b) a conservative over-approximation of the potential side-effects of the method. Both (a) and (b) depend only on the formal parameters of the method and the fields that it uses (assuming, without loss of generality, that there are no global variables in the program). This allows the information computed by the analysis for a procedure p to be used when analyzing the invocations of p.

It can be shown that our method is also applicable with partially specified post-conditions in order to estimate the potential side-effects, thereby strengthening the provided post-conditions. The details are beyond the scope of this paper.

Technically, we employ a theorem prover to compute safely the initial abstract value (which corresponds to the procedure's pre-condition) and to calculate the abstract effects of procedure calls in the most precise way [11]. This also allows us to avoid the need to compute the potential side-effects on abstract fields (called *instrumentation relation* in TVLA jargon).

Object oriented programs contain several challenging features, including intensive usage of the heap and the ability to redefine behavior using inheritance. In this paper, we address the problem of analyzing the heap, and rely on existing methods to handle inheritance, e.g., [1]. We make a simplifying assumption of working on single-threaded programs.

2 Overview

This section provides a semi-technical overview of our method for automatic assume/guarantee reasoning for programs that manipulate heap-allocated data structures.

The Process. In this paper, we restrict our attention to non-recursive procedures. This allows us to analyze procedures before they are used. Every procedure is analyzed once on an abstract value that represents a superset of the stores fulfilling the precondition. The outcome of the analysis is a safe approximation to the post-condition including the potential side-effects of every procedure. The analysis can also verify safety properties, such as the absence of memory leaks and null dereferences, and user-defined assertions, including post-conditions. The analysis of every procedure uses the (computed) post-conditions of invoked procedures.

Symbolic Operations. To realize the process described above, we implemented two operations with the use of a theorem prover: $assert$ and $assume$. The $assert$ operation takes a formula in first-order logic with transitive closure, and an abstract value, and returns TRUE if all concrete stores represented by the abstract value satisfy the formula. The $assume$ operation takes a formula in first-order logic with transitive closure and an abstract value. It returns an abstract value that

[1] Recursive calls can be handled by employing existing interprocedural shape-analysis algorithms, e.g., [8,4].

```
typedef struct list{
  int d;
  struct list *n;
} * List;

List create(int k)
@requires TRUE
{
  List p, q;
  p = q = NULL;
  while (k--) {
    p = (List) malloc(
        sizeof(struct list));
    p->n = q;
    q = p;
  }
  return q;
}
```

```
List append(List a, List b)
@requires Acyclic(a)
{
  List d = a;
  if (d == NULL) return b;
  while (d->n != NULL) {
    d = d->n;
  }
  d->n = b;
  return a;
}
void main()
@requires TRUE
{
  List x, y, z, t, s;
  x = create(I);
  y = create(J);
  z = create(K);
  P1: t = append(x, y);
  P2: s = append(t, z);
}
```

Fig. 1. A running example program that allocates 3 disjoint acyclic singly-linked lists, and calls `append` twice to concatenate the lists. I, J, and K are integers.

refines the input abstract value by eliminating concrete stores that do not satisfy the formula. [2]

Running Example. Fig. 1 shows a simple example of a C program that manipulates linked lists. We will show that our analysis is able to establish the absence of memory leaks and null dereferences in this program while analyzing every procedure once. We first analyze `append` and `create`, which are the leaf procedures. Then we analyze the `main` procedure using the results of the analyses of `create` and `append`.

Analyzing `append`. We start by analyzing the procedure `append`. Our method generates an abstract value that conservatively represents all stores that satisfy the precondition of `append`. The precondition of `append` requires the first list argument to be an acyclic list that is pointed to by a. Therefore, the $assume$ operation generates a set of shape graphs, denoted by a_1, that describes all concrete stores that contain two singly-linked lists, pointed to by a and b, where the first list is acyclic.

The abstract value a_1 contains several *shape graphs*, one of which, denoted by S_1, is shown in Fig. 2; the others are degenerate cases of S_1, where the list pointed to by a or b is empty or contains only one element. In the rest of the examples, we ignore the degenerate cases. The nodes u_1 and u_3 in S_1 represent the locations pointed to by a and b, respectively. The "summary" nodes u_2 and u_4, depicted by double circles, represent all location in the tails of the respective lists. Nodes

[2] The precision of our analysis depends on the ability of $assume$ to eliminate sufficiently many such stores; however, $assume$ need not generate the *least* abstract value. In many cases, this problem is undecidable or cannot be solve efficiently.

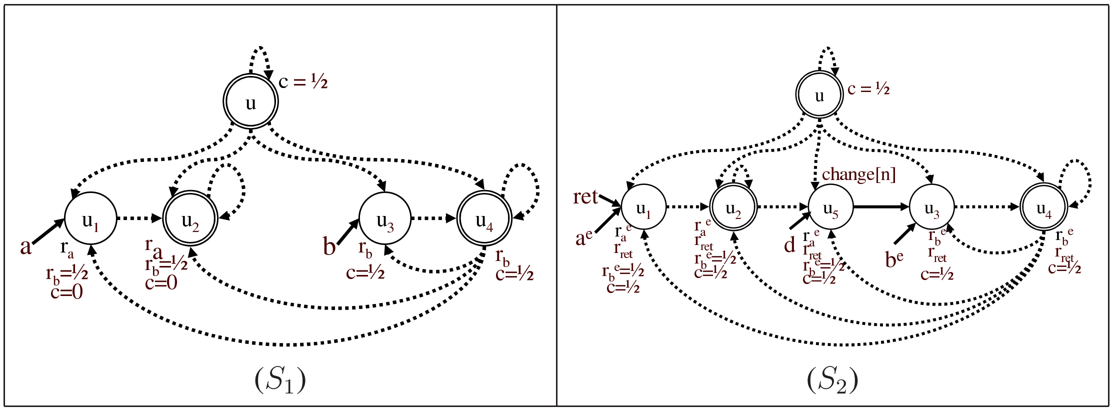

Fig. 2. (S_1) one of the shape graphs generated by $assume$(pre_$append$); (S_2) one of the shape graphs in the result of the analysis of `append`. The relations a^e, b^e, r_a^e, and r_b^e record the "entry" values of the corresponding relations. The summary node u denotes the rest of the store, i.e., nodes not reachable from a and b.

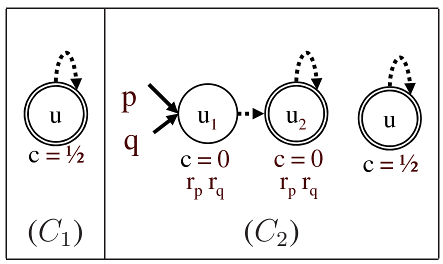

Fig. 3. (C_1) shape graph that represents all non-empty stores, generated by $assume$(pre_$create$). (C_2) one of the shape graphs generated for `create`, which contains an acyclic (newly allocated) list of length two or more, pointed to by q and p.

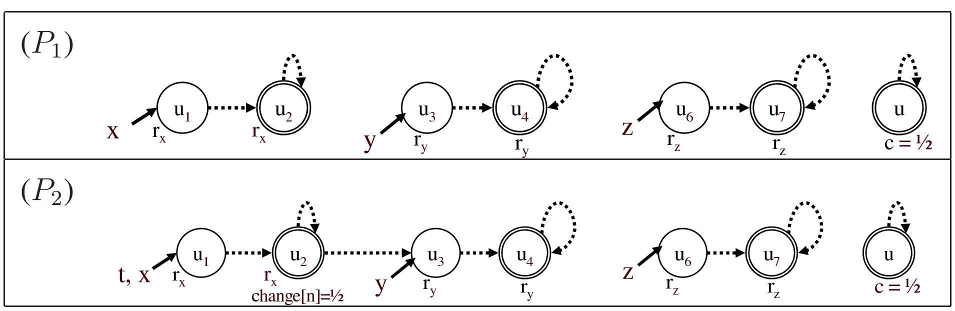

Fig. 4. (P_1) one of the shape graphs before the call to = `append(x,y);` (P_2) one of the shape graphs in the result of $assume$(post_$append$) on P_1. All nodes in P_1 and P_2 except u are marked with "$c = 0$".

marked with r_a and r_b represent locations reachable from a and b. Note that the nodes u_1 and u_2 in S_1 are marked with "$c = 0$": this means that in all concrete stores represented by S_1, the list pointed to by a is acyclic, as required by the precondition of append. The dotted edge from node u to node u_2, for example, mean that some locations represented by u may have n-fields that point to some locations represented by u_2. We do not require in the precondition that the lists be unshared (i.e., only contain nodes pointed to by at most one field); therefore, the dotted edges from u_4 to u_1 and u_2 represent possible n-fields between some of the locations represented by these nodes. Note that u_1 and u_2 are marked with "$r_b = 1/2$" meaning that the locations represented by these nodes *may be* reachable from b.

The summary node u denotes the rest of the store, i.e., nodes not reachable from a and b. The soundness of our analysis in the presence of complex heap-aliasing in the calling context is ensured by (conservatively) representing all heap cells. However, we do not represent the stack contexts—that is, the invisible variables—because they cannot be modified by the procedure.

We use abstract interpretation to analyze the code of `append`. The conservative shape analysis we use is capable of demonstrating the absence of memory leaks and null dereferences in `append`, and generates an abstract value a_2 that describes all concrete stores that may arise at the exit of `append`. One of the shape graphs in a_2, denoted by S_2, is shown in Fig. 2 (the others are degenerate cases of S_2). The analysis detects that *only* the last location of the list pointed to by a is modified, and that this location now has an n-field to the node that b points to on exit; other locations reachable from the formal parameters of append are not modified.

From each shape graph in a_2, we remove the information about the local variable d of `append`, because it is invisible to the caller of `append`, procedure `main`. For S_2 from Fig. 2, it means just "erasing" d from node u_5; in general, this may cause abstract nodes to be merged together. The resulting set of shape graphs, a_3, contains only the formal parameters, n-fields, the values recorded on entry to the procedure, and instrumentation relations. To keep track of the correlation between the values of the n fields on entry and exit, the abstraction keeps track, for each location, whether its n field was modified. This is depicted in S_2 by marking the node u_5, which corresponds to the last location of a, marked with "$change[n]$" (explained in Sec. 4.1).

The abstract value a_3 can be encoded as an equivalent logical formula [10], which precisely describes all stores represented by a_3. This formula is a post-condition for `append`. Post-condition formulas differ from pre-condition formulas since they relate the stores on entry to the procedure to the stores on exit. Technically, this is expressed using special auxiliary relations p^{entry} which refers to the value of the relation p on entry to the procedure (similar to the keyword old in Eiffel or JML).

Notice that this analysis establishes the absence of memory leaks and null dereferences in `append` not only for this program, but for all programs in which the precondition of `append` is satisfied. The reason is that we conservatively analyzed `append` with an input abstract value that describes all stores that satisfy its precondition. Also, the post-condition that has been generated describes the behavior of `append` in all such programs.

Analyzing `create`. The precondition of `create` is TRUE, thus our method generates an abstract value containing the shape graph C_1, shown in Fig. 3, and another shape graph, which represents an empty store. C_1 describes all stores with at least one location, in which n-fields have unknown values, denoted by the dotted edge on the node $u \in C_1$. The shape analysis we use establishes the absence of memory leaks and null dereferences in `create`, and generates an abstract value that contains the shape graph C_2 shown in Fig. 3. C_2 represents all stores in which q, the return value of `create`, points to an acyclic unshared list of length at least two. We remove local variable p from C_2 (as well as from the other shape graphs generated at the exit of create), and encode the resulting abstract values as an equivalent formula, which is the post-condition for `create`.

Analyzing `main`. Intraprocedural statements are analyzed as usual, and procedure calls are interpreted by checking the precondition and assuming the post-condition. A crucial issue is that we want to maintain information about the store before the call while allowing the procedure to mutate parts of the store.

The precondition of procedure `main` is TRUE; thus, analysis of `main` starts from an abstract value that describes all possible concrete stores. We skip the description of the analysis of the three calls to `create`, the result of which (denoted by P_1) is shown in Fig. 4. P_1 represents all stores in which x, y, and z point to disjoint, acyclic, unshared lists of length at least two. We check, using the $assert$ operation, that the precondition of `append` holds on all stores represented by P_1.

Instead of analyzing the code of `append` at each call, we use the $assume$ operation to generate an abstract value P_4 from the post-condition of `append`, which we generated in the previous step. This post-condition contains information about which n fields are modified (and how), and which n fields are *not* modified. This information is used by the $assume$ operation to figure out that in all stores generated by `t = append(x,y)` the list pointed to by t is acyclic, because the input lists pointed to by x and y are acyclic before the call, and all locations are unmodified, except the last location of a, which does not introduce cyclicity because the lists are disjoint. This shows that the precondition for the next call to `r = append(t,z)` holds. From this fact, we conclude that there are no memory leaks and null dereferences in `main`.

Technically, calls are handled using the standard notion of a two-vocabulary store, which relates the store before and after the call to a procedure. The two-vocabulary store contains two versions of each binary relation symbol, where the unprimed version describes the store before the call, and the primed version describes the store after the call. The two-vocabulary store is used "locally" only during the modelling of the call; the primed relations are not involved in the analysis of other program statements.

This example shows that establishing simple properties of a caller, such as absence of memory leaks, requires establishing stronger properties about the callee, such as acyclicity. The analysis is conservative and thus we may fail to establish certain properties of the program even if they hold. However, the analysis can be rather precise; in particular, it may compute a post-condition that is stronger than a post-condition that a user would provide.

3 Preliminaries

The method described in this paper is based on the the shape-analysis framework [9], implemented in the TVLA system (**T**hree-**V**alued-**L**ogic **A**nalyzer) [6].

3.1 Stores as Logical Structures

In this approach, concrete memory configurations or *stores* are encoded as logical structures (associated with a *vocabulary* of relation symbols with given arities) in terms of a fixed collection of *core relations*. Core relations are part of the underlying semantics of the language to be analyzed; they record atomic properties of stores.

For instance, to represent the stores manipulated by programs that use type `List` (declared in Fig. 1), we use the relation $q(v)$ to denote whether a pointer variable `q` points to memory cell v, and $n(v_1, v_2)$ to denote whether the n-field of v_1 points to v_2.

2-valued logical structures then represent memory configurations: the individuals are the set of memory cells; a nullary relation represents a Boolean variable of the program; a unary relation represents either a pointer variable or a Boolean-valued field of a record; and a binary relation represents a pointer field of a record. There are also integrity constraints, which capture the semantic requirements of logical structures that represent stores (e.g., a binary relation that represents a pointer field must be a partial function).

We can model dynamic memory allocation using an additional unary relation $active(v)$. Conceptually, we assume that the number of nodes in a structure is always infinite; $active$ is set to FALSE for all nodes in an empty heap. Allocation is modelled by setting $active$ to TRUE for the newly allocated node. To simplify the exposition, we omit the $active$ relation in this paper.

A set of stores is then represented by a (finite) set of 3-valued logical structures. The abstraction is defined using an equivalence relation on individuals, and considering the (finite) quotient structure with respect to this equivalence relation; in particular, each individual of a 2-valued logical structure (representing a concrete memory cell) is mapped to an individual of a 3-valued logical structure according to the vector of values that the concrete individual has for a user-chosen collection of unary abstraction relations. Other relations are collapsed accordingly. Canonical abstraction ensures that each 3-valued structure is no larger than some fixed size, known *a priori*.

The following graphical notation is used for depicting 3-valued logical structures:

- Individuals are represented by circles containing their names and values for unary relations (0 values are usually omitted).
- A summary individual is represented by a double circle.
- A unary relation p corresponding to a pointer-valued program variable is represented by a solid arrow from p to the individual u for which $\iota(p)(u) = 1$, and by the absence of a p-arrow to each node u' for which $\iota(p)(u') = 0$.
- A binary relation q is represented by a solid arrow labeled q between each pair of individuals u_i and u_j for which $\iota(q)(u_i, u_j) = 1$, and by the absence of a q-arrow between pairs u_i' and u_j' for which $\iota(q)(u_i', u_j') = 0$.
- Relations with value $1/2$ are represented by dotted arrows.

3.2 Instrumentation Relations

The abstraction function on which an analysis is based, and hence the precision of the analysis defined, can be tuned by (i) choosing to equip structures with additional *instrumentation relations* to record derived properties, and (ii) varying which of the unary core and unary instrumentation relations are used as the set of abstraction relations. Each instrumentation relation symbol p of arity k is defined by an *instrumentation-relation definition formula* $\psi_p(v_1, \ldots, v_k)$, with k free variables. Instrumentation relation symbols may appear in the defining formulas of other instrumentation relations as long as there are no circular dependences.

The introduction of unary instrumentation relations that are then used as abstraction relations provides a way to control which concrete individuals are merged together into an abstract individual, and thereby control the amount of information lost by abstraction. Instrumentation relations that involve reachability properties, which can be defined using the * operator (i.e., transitive closure), often play a crucial role in the definitions of abstractions. For instance, reachability properties from specific pointer variables have the effect of keeping disjoint sublists summarized separately. We use the following instrumentation relations for the analysis of programs that use type `List`. The relation $r_{x,n}(v)$ denotes that v is reachable from pointer variable `x` along `n` fields; it is defined by $\exists\, v_1 \colon x(v_1) \wedge n^*(v_1, v)$. The relation $c_n(v)$ means that v is on a directed cycle of `n` fields; it is defined by $\exists\, v_1 \colon n(v_1, v) \wedge n^*(v, v_1)$. (The relation symbols $r_{x,n}$ and c_n were abbreviated as r_x and c in the figures shown in Sec. 2.)

From the standpoint of the concrete semantics, instrumentation relations represent cached information that could always be recomputed by reevaluating the instrumentation relation's defining formula in the local state. From the standpoint of the abstract semantics, it refines the abstraction by keeping certain information precise, whereas reevaluating the instrumentation relations's defining formula in

	$pre \leftarrow formals2actuals(\text{pre\_}f)$
$a_1 \leftarrow assume[\text{pre\_}f]$	$assert[pre](p_1)$
$a_1^{entry} \leftarrow record\_entry(a_1)$	$post \leftarrow formals2actuals(\text{post\_}f)$
$a_2 \leftarrow analyze[f](a_1^{entry})$	$post2 \leftarrow rename(post)$
$a_3 \leftarrow project[f](a_2)$	$p_2 \leftarrow primed2unknown(p_1)$
$\text{post\_}f \leftarrow \hat{\gamma}(a_3)$	$p_3 \leftarrow assume[post2](p_2)$
	$p_4 \leftarrow primed2unprimed(p_3)$
(a)	(b)

Fig. 5. (a) The analysis of a procedure f; (b) A call to the procedure f.

the local (3-valued) structure would lead to a less precise value.

4 Bottom-up Assume-Guarantee Reasoning

Specifications We use pre_f and post_f to denote the pre- and post-conditions of a procedure f. In our method, pre_f and post_f are formulas in first-order logic with transitive closure over a vocabulary that describes formal parameters and visible fields of the procedure f. The precondition is specified by the user, and the post-condition is generated by our system.

Example 4.1 *In the running example, the precondition of* `append` *can be expressed by: pre_append* $\stackrel{\text{def}}{=} \forall v : r_{a,n}(v) \Rightarrow \neg(\exists w : n(v,w) \wedge n^*(w,v))$

The analysis of a procedure f consists of the operations shown in Fig. 5(a). First, we use the $assume$ algorithm described in [11] to generate an abstraction of all concrete stores that satisfy pre_f. Because each procedure is analyzed separately, we can use different abstractions for each procedure. For canonical abstraction, this means it is possible to use different sets of relation symbols during the analysis of each procedure: we only use those relations that denote "visible" variables and fields.

Example 4.2 *In our example, we do not represent the variables x, y, z of* `main` *while analyzing $append$. This way, we can use a more coarse representation of those data-structures that do not change during the execution of the procedure.*

The next operation, $record\_entry$, records the information about the store on entry to f, using auxiliary relation symbols. This information is used later (i) to express the post-condition of f, which refers to the values of relations on entry, and (ii) to record information that distinguishes between different nodes in the shape graph, whenever this information can be expressed in terms of f's formal parameters. The latter prevents loss of important information about the stores before the call to f.

Example 4.3 *For the analysis of* `append`*, we introduce additional relation symbols, denoted a^{entry}, b^{entry}, $r_{a,n}^{entry}$, $r_{b,n}^{entry}$, $c[n]^{entry}$, n^{entry}. Note that these are* not *instrumentation relations.*

The operation $analyze(a_1^{entry})$ performs abstract interpretation of f, and generates a set of shape graphs a_2 that represent a superset of all concrete stores that may arise at the exit of f.

The next operation, $project[f](a_2)$, (i) generates a sub-structure of a_2 by taking the restriction of a_2 to relations that denote formal parameters, "entry" information, and instrumentation relations; and (ii) applies canonical abstraction to the restriction; the result is denoted by a_3.

Finally, we generate from a_3 a formula $\hat{\gamma}(a_3)$ [10] such that a concrete store satisfies $\hat{\gamma}(a_3)$ if and only if it is represented by a_3 (and satisfies all integrity constraints). $\hat{\gamma}(a_3)$ is a post-condition of f.

4.1 $change[n]$ Instrumentation

The abstraction that we use causes significant loss of precision for binary information, if no special instrumentation relations are introduced. Merely recording values of the binary relation n on entry is insufficient.

For example, the value of n on $\langle u_4, u_4 \rangle$ in S_1 in Fig. 2 is $1/2$, meaning that some of the locations represented by u_4 have n fields that point to a location also represented by u_4. The operation $record\_entry$ initializes n^{entry} to the same values as n, but under abstraction this correlation is lost: the fact that n^{entry} and n have the same value $1/2$ on $\langle u_4, u_4 \rangle$ means that some locations have n^{entry} fields, but not necessarily that the same locations have n fields.

To address this problem, we keep the track of locations for which the n-field is *not* modified, using the instrumentation relation $change[n](v)$ defined by: $change[n](v) \stackrel{\text{def}}{=} \neg\forall w : (n^{entry}(v, w) \iff n(v, w))$. The operation $record\_entry$ initializes $change[n]$ to 0 for all nodes. The TVLA system automatically updates the value of $change[n]$ (and other instrumentation relations) during the analysis via differencing [7]. Note that the $project$ operation does not remove $change[n]$ because it is an instrumentation relation that is expressed in terms of variables that are visible in the caller. Thus, $change[n]$ becomes a part of the postcondition.

Conceptually, our method can handle any number of fields, by generating $change[p]$ instrumentation relation for each field p. In practice, this affects the ability of the theorem prover to discharge the queries posed by the analysis.

In our example, $change[n]$ is 0 for all nodes reachable from b; it implies that the n-fields in the list pointed to by b are not modified after the first call to `append`. As shown in the next section, this information is crucial for establishing that the result of the first call to `append` is acyclic, as required by the precondition of the second call to `append`.

4.2 Procedure Call

A call to procedure f is replaced by the operations shown in Fig. 5(b): We replace the formal parameters in pre_f by the actual arguments passed to f; this operation is called $formals2actuals$. Note that it is not a renaming of variables, but rather a renaming of relation symbols. We check that the precondition of f holds for all concrete stores before the call, using $assert$. The subsequent operations create from p_1 a set of shape graphs p_4 that represents a superset of all stores after the call to $append$.

Example 4.4 *We replace each occurrence of a in pre_$append$ by x to get the formula pre defined by $r_{x,n}(v) \Rightarrow \neg(\exists w : n(v,w) \wedge n^*(w,v))$. The set of shape graphs p_1 represents all concrete stores that may arise at program location $P1$ in $main$. The shape graph p_1 shown in Fig. 4 represents three disjoint acyclic unshared singly-linked lists, pointed to by x, y, and z. The precondition of* `append` *holds for p_1.*

We use the standard notion of a two-vocabulary store, which relates the store before and after the call to a procedure. The two-vocabulary store contains two versions of each binary relation symbol, where the unprimed version describes the store before the call and the primed version describes the store after the call. We do not introduce primed versions of unary relation symbols, because these relation symbols denote program variables of the caller, whose values cannot be changed by the procedure call, as they are invisible to the callee. One exception is the return value of a procedure, which is described by a primed version of the corresponding relation symbol. [3]

Remark. It is important to distinguish between the two-vocabulary store (primed and unprimed versions of relations), and the "entry" relations. In our example, "entry" relations record the values of formal parameters on entry to `main`, i.e., the calling procedure, and are used at the exit of `main` to generate its post-condition. We do not introduce primed versions for the entry relations, because they cannot be changed by the call to `append`.

Two-vocabulary stores are used "locally" only during the modeling of the call; the primed relations are not involved in the standard fixpoint computation. Again, this is important because our analysis can be exponential in the number of relations.

Next, we replace the formal parameters in post_f by the actual arguments passed to f using $formals2actuals$; the result is a new formula denoted by $post$. Note that the formula $post$ may contain "entry" relations, which constrain the store before the call, and (unprimed) relations, which constrain the store after the call. Therefore, we use the operation $rename$ to rename each (unprimed) non-"entry"

[3] Alternatively, we can define a primed version for each relation that denotes a program variable; the precondition would contain $\forall v : x(v) \iff x'(v)$ for all formal and invisible program variables. This solution is undesirable because our analysis can be exponential in the number of variables.

relation symbol in $post$ by its primed version, and then replace all "entry" relations to their unprimed versions. The result is a formula $post2$ that uses two vocabularies.

Example 4.5 *For example, suppose that the post formula for the first call to* `append` *is* $\forall v : (r^{entry}_{x,n}](v) \lor r^{entry}_{y,n}](v)) \Rightarrow (r_{t,n}(v))$*, then* $post2$ *is the formula* $\forall v : (r_{x,n}(v) \lor r_{y,n}(v)) \Rightarrow (r_{t',n'}(v))$.

The overall effect is that all occurrences of a^{entry}*,* b^{entry}*, and* n^{entry} *in post_append are replaced by* x*,* y*, and* n *respectively. The occurrences of* t *and* n outside *without "entry" superscripts are replaced by* t' *and* n'*.*

To evaluate the two-vocabulary formula $post2$ on p_1, we need to extend p_1 with primed version of relations, first initialized to unknown; this operation is called $primed2unknown$, and results in a new set of shape graphs, p_2. Then, we apply the symbolic operation $assume$, which generates from p_2 a set of shape graphs p_3. The assume operation refines the primed relations in p_2 to more precise values, by excluding values that do not happen in any valid store.

Note that each instrumentation relation also has a primed version, with a primed version of the defining formula. For example, the instrumentation relation $r_{x,n}(v)$ defined by $\exists w : x(w) \land n^*(w, v)$ has a primed version $r'_{x,n}$ defined by $\exists w : x(w) \land n'^*(w, v)$, with x and not x' because x has no primed version for this call.

The $assume$ operation uses a theorem prover to infer these values from the definitions of the primed instrumentation relations and from the values of the primed core relations. This includes reachability instrumentation, $change[n]$. It can be shown that abstract fields (so-called "ghost-fields") can be treated in a similar way to instrumentation relations; we omit this from the paper for the reasons of space. This provides a neat solution to the problems of updating reachability and abstract fields without breaking abstraction layers!

Finally, after $assume$ is completed, we can discard the unprimed values and return to a single-vocabulary structure; this operation is called $primed2unprimed$, it copies the values of the primed relations to the corresponding unprimed relations.

4.3 Evaluation of the Method

To evaluate the feasibility of the method, we implemented our method using the TVLA system [6] and the Simplify theorem prover [3]. We established the correctness of the running example of this paper using TVLA enhanced by a symbolic engine to compute $assume$ and $assert$. We used the Simplify theorem prover to discharge queries posed by the symbolic part. We modeled transitive closure using a binary relation and a set of simple axioms. Currently, we are developing a Java front end, and plan to perform more experiments in which we apply the approach to Java methods.

References

[1] M. Barnett, R. DeLine, K. R. M. Leino M. Fhndrich, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 2004.

[2] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM, 1979.

[3] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003. http://research.compaq.com/SRC/esc/Simplify.html.

[4] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv M. A relational approach to interprocedural shape analysis. In *SAS*, Lecture Notes in Computer Science. Springer, 2004.

[5] K. Rustan M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *PLDI*, pages 246–257, 2002.

[6] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *SAS'00, Static Analysis Symposium*. Springer, 2000. http://www.cs.tau.ac.il/$\sim$ tvla.

[7] T. Reps, M. Sagiv, and A. Loginov and. Finite differencing of logical formulas for static analysis. In *ESOP*, pages 380–398, 2003.

[8] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. *Lecture Notes in Computer Science*, 2027:133–149, 2001.

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

[10] G. Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2003. Available at "http://www.cs.tau.ac.il/$\sim$ gretay".

[11] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, pages 530–545, 2004.