Call invariants

Shuvendu K. Lahiri and Shaz Qadeer

Microsoft Research

Abstract. Program verifiers based on first-order theorem provers model the program heap as a collection of mutable maps. In such verifiers, preserving unmodified facts about the heap across procedure calls is difficult because of scoping and modification of possibly unbounded set of heap locations. Existing approaches to deal with this problem are either too imprecise, require introducing untrusted assumptions in the verifier, or resort to unpredictable reasoning using quantifiers. In this work, we propose a new approach to solve this problem. The centerpiece of our approach is the *call invariant*, a new annotation for procedure calls. A *call invariant* allows the user to specify at a call site an assertion that is inductively preserved across an arbitrary update to a heap location modified in the call. Our approach allows us to leverage existing techniques for reasoning about call-free programs to precisely and predictably reason about programs with procedure calls. We have implemented the approach and applied it to the verification of examples containing dynamic memory allocations, linked lists, and arrays. We observe that most call invariants have a fairly simple shape and discuss ways to reduce the annotation overhead.

1 Introduction

Floyd-Hoare logic is a framework for decomposing the partial correctness checking of a program into smaller proof obligations, where a Floyd-Hoare triple $\{P\}\ s\ \{Q\}$ is associated with each statement s in the program [10]. Verification condition (VC) generation based on Dijkstra's *weakest liberal precondition* (*wp*) predicate transformer allows precise reasoning about Floyd-Hoare triples without requiring intermediate assertions for loop-free and call-free statements [7]. The use of automated theorem provers (including *satisfiability modulo theories* (SMT) solvers [19]) for checking the verification conditions provide a scalable and precise approach to program verification, and forms the basis of several tools (e.g. `ESC/Java` [8], `Spec#` [5], `HAVOC` [12]).

However, this framework is not as effective in the presence of the heap and procedure calls. The main issue is to preserve unmodified facts about the part of the heap in the caller's scope that is not in scope of the callee. More formally, a procedure specification comprises of (a) preconditions, (b) postconditions and (c) set of variables modified by the procedure. Since the heap is modeled as a collection of maps, a procedure that modifies a location in a map has to specify that the entire map is potentially modified. The only facts related to such a

modified map, known after a procedure call has to come from the postconditions of the callee procedure. However, the postconditions can only refer to part of the map *in scope*, i.e. the locations reachable from globals and parameters. This means that unmodified facts at caller's scope about a modified map may not be preserved across a procedure call. Matters are further complicated as a procedure call might update an unbounded number of locations in a map. Efficiently decidable SMT-based logics (e.g. linked lists [12], arrays [6]) that deal with a bounded number of heap updates (for a loop-free, call-free program fragment) are rendered ineffective in the presence of the unbounded number of updates.

An existing approach to address this problem has been to introduce *frame axioms* to allow preserving certain unmodified facts [13]. These axioms are not verified in the same spirit as the rest of the user annotations, and may introduce unsoundness in the verifier. Moreover, these frame axioms are encoded using complex quantified facts in the verification condition that severely compromises the predictability of the underlying theorem prover. Verification of such quantified formulas require expert users to be able to guide the theorem provers. These shortcomings make *wp* based Floyd-Hoare reasoning less appealing for reasoning about programs with scoping and the heap.

In this work, we present an alternative approach based on the following insight:

> For any statement s in the program, if an assertion R is preserved by an abstract (re-)execution of s in which the heap locations modified by s are updated nondeterministically, then R is preserved across the statement s.

We use an instance of this general rule for procedure calls to deal with the imprecision due to scoping. We also provide a new annotation called *call invariant* for the user to specify an inductive hypothesis when the abstract execution could be unbounded. Given a program with call invariant annotations, we perform a source-to-source transformation to create another program that can be reasoned with any existing technique for call-free programs. In particular, this allows a user to leverage existing *wp*-based verifiers to analyze programs with procedure calls with unbounded heap updates.

One can also view our approach as a strategy to augment the underlying first-order theorem provers with an induction scheme to verify formulas containing unbounded number of heap updates. The call invariant (provided by the user) plays the role of an inductive hypothesis and the underlying theorem prover is used to discharge the proof obligations for establishing the inductive hypothesis. However, there are several advantages of formalizing the inductive hypothesis at the program level instead of at the level of a formula:

1. The call invariants are specified as program annotations independent of the underlying prover. Therefore, the user does not need to interact with the specific syntax of the underlying theorem provers.
2. We formulate the call invariants as loop invariants. This opens the possibility to leverage existing loop invariant synthesis techniques to infer call invariants in many cases.

We have augmented the Boogie [3] verifier with call invariant annotations, and have applied it to verify a set of examples containing dynamic memory allocation, linked lists and arrays. These examples were already annotated with preconditions and postconditions — we discuss the additional annotation burden due to call invariants. We introduce useful syntactic sugars and observe the common shape of most call invariants and additional specification required to prove these examples. We also discuss tradeoffs in reducing the additional burden at the cost of slight complication of the assertion logic, without sacrificing soundness.

2 Motivation

Consider two versions of a program in Figure 1 written in a variant of the Boogie language [3]. The example is an abstraction of a real-life device driver kbdclass [20] that uses multiple lists of device extensions. The first version (on the left) has single procedure with no procedure calls, and the second version (on the right) has a procedure call.

2.1 Program without procedure calls

Let us first look at the example in Figure 1(a). Initially ignore the lines starting with **pre**, **post**, **inv** and **modifies**, which denote annotations. The example contains two map (or array) variables N and D to model two fields in an object. The procedure *Proc1* takes two pointers p and q to denote the heads of the two disjoint acyclic lists $\{\mathsf{p}, \mathsf{N}[\mathsf{p}], \mathsf{N}[\mathsf{N}[\mathsf{p}]], \ldots, \mathsf{nil}\}$ and $\{\mathsf{q}, \mathsf{N}[\mathsf{q}], \mathsf{N}[\mathsf{N}[\mathsf{q}]], \ldots, \mathsf{nil}\}$ respectively. The procedure first initializes the D field of all the pointers in the linked list starting at p in a while loop, and then non-deterministically deletes some entries from the list starting at q — this mimics removing elements from a list that satisfy some criteria. We would like to prove that the D field has been correctly initialized for the list from p.

The assertions in **pre** and **post** denote preconditions and postconditions of a procedure. The precondition states that the two lists are disjoint and acyclic: we use the set constructor $\mathsf{Btwn}(m, u, v)$, where m is a map value of type $\mathsf{int} \rightarrow \mathsf{int}$ and u and v are values of type int, to denote the set of values $\{u, m[u], m[m[u]], \ldots, v\}$ when v lies in the set, or $\{\}$ otherwise [16, 12]. The postcondition states that the value of D map at all the elements of list from p is 1. The "modifies" clause in **modifies** says that the maps N and D are modified by the procedure, possibly at all locations. Loop invariant assertions are provided using **inv** annotations. The expression **old**(x) denotes the value of a variable x at the entry to a procedure (when used in a postcondition), or at the entry of a loop (when used in a loop invariant). The loop invariants on the first loop states that the variable iter points to the list from p, and all the entries upto iter have been initialized to 1. The first loop establishes the postcondition of the procedure on exit from the loop — the problem is to preserve it across the second loop. The first loop invariant for the second loop states that the set

```
var N : int → int;
var D : int → int;

pre Btwn(N, p, nil) ∩ Btwn(N, q, nil) = {nil}
post ∀u ∈ Btwn(N, p, nil).u = nil ∨ D[u] = 1
modifies D, N
proc Proc1(p : int, q : int) : void =
  var iter : int;
  iter := p;

  inv iter ∈ Btwn(N, p, nil)
  inv ∀u : int ∈ Btwn(N, p, nil).
        u ∈ Btwn(N, iter, nil) ∨ D[u] = 1
  while (iter) do D[iter] := 1; iter := N[iter];

  iter := q;

  inv Btwn(N, p, nil) = Btwn(old(N), p, nil)
  inv iter ∈ Btwn(old(N), q, nil)
  while (iter ∧ N[iter]) do
    if (∗) N[iter] := N[N[iter]];
    iter := N[iter];
```

```
var N : int → int;
var D : int → int;

pre Btwn(N, p, nil) ∩ Btwn(N, q, nil) = {nil}
post ∀u ∈ Btwn(N, p, nil). u = nil ∨ D[u] = 1
modifies D, N
proc Proc1(p : int, q : int) : void =
  var iter : int;
  iter := p;

  //Loop invariants omitted
  while (iter) do D[iter] := 1; iter := N[iter];

  cinv cframe(Btwn(N, p, nil))
  call Proc2(q);

updates N @ Btwn(N, t, nil)
proc Proc2(t : int) : void =
  var iter : int;
  iter := t;

  inv iter ∈ Btwn(old(N), t, nil)
  while (iter ∧ N[iter]) do
    if (∗) N[iter] := N[N[iter]];
    iter := N[iter];
```

Fig. 1. Example with (a) no procedure calls and (b) a procedure call.

of pointers in the list from p remains unchanged. The second loop invariant says that the iterator variable iter points to elements in the list from q.

These annotations are sufficient to prove the postcondition, since the map D does not change in the second loop. The annotated program can be encoded precisely in the assertion logic if the logic is closed under weakest (liberal) precondition [7] of statements in the programming language. Such logics with decision procedures have been proposed in [12], thereby providing an algorithm for checking such annotated programs.

2.2 Program with procedure calls

Now let us look the second version in Figure 1(b), where the second loop has been moved to a procedure *Proc2*. Let us initially ignore the annotation in **cinv**, and the updates N @ . annotation on *Proc2*. Instead, let us pretend that we only have a **modifies** N annotation for *Proc2*. Since *Proc2* modifies the map N, any fact involving N will be invalidated after the call to *Proc2*. Therefore, the postcondition of *Proc1* will not be provable. It is not hard to see that we cannot write any specification about the heap in the scope of *Proc2* (namely the pointers

in the list reachable from t) that would allow us to prove the postcondition for *Proc1*.

One approach to address the imprecision has been to use *frame axioms* that allow the user to specify how to preserve certain unmodified facts [13]. However, the use of frame axioms can lead to unsoundness as they are not verified. Besides, these frame axioms have complex quantified structure that may destroy the predictability of the underlying theorem provers. In the rest of this section, we show how our approach helps retain precision in the presence of procedure calls, without requiring the use of frame axioms.

First, let us look at the new annotation on *Proc2*. The annotation updates X @ $\mathcal{S}$ denotes that the map X could have been modified only at locations in $\mathcal{S}$ by the procedure., where the set-expression $\mathcal{S}$ is interpreted at entry to a procedure. This annotation is actually a syntactic sugar for a particular postcondition that we explain in Section 3.2, and does not introduce any new annotation construct. In this example, the annotation is used to specify that the map N is only modified in the locations present in the list from t at the start of the procedure.

Second, we introduce a new annotation construct called *call invariants* (using **cinv**) that allows the user to annotate a call site of a procedure. A user can specify an assertion R inside **cinv** at a call site of a procedure — with the intention that R is preserved across modifications to the maps in the callee. We use the syntactic sugar cframe(e) to denote the assertion that the value of the expression e is preserved across the call. In this example, the assertion in **cinv** states that the set of pointers in the list from p is the preserved across the call to *Proc2*.

These annotations suffice to prove the postcondition of *Proc1*. Indeed, we can prove the specifications of this example (including the new proof obligations for showing that assertions in **cinv** are really preserved across a call). Not only that, the proof obligations can be encoded using the same logic that was used to prove the example without a procedure call in Figure 1(a).

3 Call invariants

3.1 Source and assertion language

Figure 2 shows a simple programming language. The language supports scalar and map variables (*Scalars* and *Maps* respectively) and various operations on them. Let $Vars = Scalars \cup Maps$. The type of any variable $\mathsf{x} \in Scalars$ is integer (int), and the type of any variable $\mathsf{X} \in Maps$ is a map from integers to integers ($\mathsf{int} \rightarrow \mathsf{int}$). The standard assignment statement for scalars is extended with assignment statements for maps. The statement for variable introduction var x in s endvar introduces a variable x with an arbitrary value in s (the variable introduction rule for a map variable is similar). The statement **assert** ϕ behaves as a skip when the formula ϕ evaluates to true in the current state; else the execution of the program *fails*. The statement **assume** ϕ behaves as a skip when the formula ϕ evaluates to true in the current state; else the execution

$$
\begin{array}{lll}
\mathsf{x}, \mathsf{y} & \in \mathit{Scalars} & \\
\mathsf{X}, \mathsf{Y} & \in \mathit{Maps} & \\
e & \in \mathit{Expr} & ::= \mathsf{x} \mid c \mid e \pm e \mid \mathsf{X}[e] \\
s, t & \in \mathit{Stmt} & ::= \mathsf{skip} \mid \textbf{assert}\ \phi \mid \textbf{assume}\ \phi \mid \mathsf{x} := e \mid \mathsf{X} := \mathsf{Y} \mid \mathsf{X}[e] := e \mid \\
 & & \quad \mathsf{var}\ \mathsf{x}\ \mathsf{in}\ s\ \mathsf{endvar} \mid \mathsf{var}\ \mathsf{X}\ \mathsf{in}\ s\ \mathsf{endvar} \mid s; s \mid \mathsf{x} := \mathsf{call}\ f(e) \mid \\
 & & \quad \mathsf{if}\ (e)\ \mathsf{then}\ s\ \mathsf{else}\ t \mid \mathsf{while}\ (e)\ \mathsf{do}\ s \\
P, Q, R, \phi, \psi & \in \mathit{Formula} & ::= e \leq e \mid \phi \wedge \phi \mid \neg\phi \mid e \in \mathcal{S} \mid \mathcal{S} \subseteq \mathcal{S} \mid \ \ldots \\
\mathcal{S} & \in \mathit{SetExpr} & ::= \mathsf{Btwn}(\mathsf{X}, e, e) \mid \mathsf{Inverse}(\mathsf{X}, e) \mid [e, e) \mid \ \ldots \mid \mathcal{S} \cup \mathcal{S} \mid \mathcal{S} \setminus \mathcal{S} \mid \ \ldots
\end{array}
$$

Fig. 2. A simple programming language and assertion logic.

of the program is *blocked*. Expression terms of the statements and formulas are denoted by *Expr*, and include scalar variables, constants, arithmetic expressions and map lookups. The language also supports sequential composition, procedure calls, conditional statements and while loops. Allocation and deallocation can be modeled by introducing a special map Alloc to track the allocation status of objects, but is not built into the language. We show an example of dymamic allocation in the next section 4.1.

The formulas in *Formula* constitute the *assertion logic* for specifying contracts for programs in this language. The language of formulas in *Formula* is extensible, and includes relational, Boolean operations and set operations. *SetExpr* represent set-valued expressions and can be constructed from various set constructors such as Btwn and $[e, e)$ and other operations on sets. The weakest (liberal) precondition of an assertion $\phi \in \mathit{Formula}$ with respect to a statement $s \in \mathit{Stmt}$ is denoted as $wp(s, \phi)$. Intuitively, $wp(s, \phi)$ is a formula that represents the set of states for which executing s does not fail any assertions in s and moreover if the execution terminates, it does so in a state satisfying ϕ. Figure 6 in Appendix A shows the weakest precondition for the simple statements in our language, and are fairly standard [4]. The assertion logic in *Formula* is *closed under wp*, when for any $\phi \in \mathit{Formula}$ and $s \in \mathit{Stmt}$, $wp(s, \phi) \in \mathit{Formula}$. The following proposition relates checking partial correctness of statements using Floyd-Hoare triples [10] and provability in a logic. Let us refer to loop-free and call-free statements as *simple* statements.

Proposition 1. *If the assertion logic in Formula is closed under wp for simple statements in Stmt, then for any simple statement s, (i) the logical formula $(P \implies wp(s, Q))$ is in Formula, and (ii) is valid if and only if the Floyd-Hoare triple $\{P\}\ s\ \{Q\}$ holds.*

In such a case, an automated theorem prover for checking assertions in *Formula* provides a method (an algorithm when the theorem prover is complete and terminating) to check Floyd-Hoare triples expressed in the logic. In the presence of loops that may have unbounded updates, the user decomposes the problem by specifying a loop invariant.

However, the presence of procedure calls and heap makes reasoning in Floyd-Hoare logic imprecise because it introduces the challenge of preserving unmodified facts about the heap in the callers scope (and not in the callee's scope) across (a possibly unbounded) update to the heap in the callee.

3.2 Call invariant and program instrumentation

First, for each global map $\mathsf{X} \in \mathit{Maps}$, we introduce a state variable $\mathsf{MS_X}$, whose interpretation is a set of locations. Intuitively, the value of $\mathsf{MS_X}$ at exit from a procedure captures the set of locations where X was modified between the entry and exit to the procedure. The source program is automatically instrumented to update the $\mathsf{MS_X}$ variables as follows:

- For any explicit update to the map X, $\mathsf{X}[e_1] := e_2;$, we insert $\mathsf{MS_X} := \mathsf{MS_X} \cup \{e_1\}$; before the update to X.
- Every procedure has a precondition **pre** $\mathsf{MS_X} = \{\}$.
- Before any procedure call that has a **modifies** X annotation, we save the current value of $\mathsf{MS_X}$ into a caller local variable, set $\mathsf{MS_X}$ to $\{\}$. Upon return from the procedure, we union the saved set with the value of the set after the procedure call.

It is not hard to see that the value of $\mathsf{MS_X}$ at exit from a procedure captures the set of locations where X was modified between the entry and exit to the procedure. For this instrumented program, the user can specify preconditions, postconditions and loop invariants in terms of $\mathsf{MS_X}$ variable just as any other state variable.

Next, we introduce an annotation construct called *call invariant* at the call site of a procedure specified using **cinv** R, where $R \in \mathit{Formula}$. For a call site that may (transitively) modify locations in the map X, we provide the following instrumentation in addition to the updates for MS_X:

```
var X_pre, X_post in
  X_pre := X;
  call Foo(e); //procedure call
  X_post := X;
  X := X_pre;

  inv R[X_pre/old(X)]
  while (*) do
    var u, v in assume u ∈ MS_X; X[u] := v; endvar

  assume X = X_post;
endvar
```

First, it copies the value of X before and after the call into local variables $\mathsf{X_{pre}}$ and $\mathsf{X_{post}}$ respectively. It restores X to the value before the call, and introduces a non-deterministic loop that updates X at one of the locations in $\mathsf{MS_X}$.

Finally, the assume relates the value of X after the loop with the value at the end of the procedure call $\mathsf{X}_{\mathsf{post}}$. The purpose of the loop is to model the abstract re-execution of the procedure call that nondeterministically modifies the heap locations modified by the callee, starting from the state before the call. The call invariant R (specified using **cinv** R) is checked as a loop invariant for this loop; any occurrence of **old**(X) is replaced with copy $\mathsf{X}_{\mathsf{pre}}$, the value of X just before the procedure call. Although we have described the instrumentation for a single map variable, our implementation allows for multiple maps and is presented in the Appendix B.

Syntactic sugars We introduce two syntactic sugars to make the specifications concise and readable:

1. We introduce $\mathsf{updates}\ \mathsf{X}\ @\ \mathcal{S}$ for a set-valued expression $\mathcal{S}$, as a syntactic sugar for the following annotations:

$$\begin{array}{l}\textbf{modifies}\ \mathsf{X}, \mathsf{MS}_{\mathsf{X}};\\ \textbf{post}\ \mathsf{MS}_{\mathsf{X}} \subseteq \textbf{old}(\mathcal{S})\end{array}$$

2. The use of **old**(X) in a call invariant R refers to the value of X prior to the procecudure call (same as $\mathsf{X}_{\mathsf{pre}}$ at the time of the call). We provide a sugar $\mathsf{cframe}(.)$ to denote that an expression is preserved by the procedure. For an expression e, $\mathsf{cframe}(\mathsf{e})$ expands to $\mathsf{e} = \textbf{old}(\mathsf{e})$. This is most useful when specifying that the value of a scalar expression or a set-valued expression is preserved.

Bounded updates Recall that the scoping problem exists even when a callee modifies a bounded number of heap locations. Our program instrumentation is still essential to preserve facts at the callers scope. However, if callee has a postcondition that bounds MS_{X} to a bounded number of locations (say n), then one does not require the user to specify a call invariant R. In such cases, it suffices to unroll the loop that modifies locations in MS_{X} n times, and eliminate the need for the call invariant.

4 Evaluation

We have built a prototype implementation of call invariant annotations over the `Boogie` program verifier [3]. In addition to specifying procedure preconditions, postconditions and loop invariants, the user can specify call invariants at call sites using the **cinv** annotation. We highlight the annotations required related to call invariants in the program in addition the already provided preconditions and postconditions. We discuss more about the **cinv .** call invariants (the call invariants that are both highlighted and underlined) later in Section 4.4.

We perform the program instrumentation to create the transformed program where the call invariants are desugared as loop invariants as described earlier.

```
var Alloc : int → bool;
var D : int → int;

updates D @ Inverse(Alloc, false)
proc Proc4() : void =
  var y : int;
  while (*) do
    y := new; D[y] := 0;
    /* Add y to a list */
```

```
proc Proc3() : void =
  var x : int;
  x := new;
  D[x] := 5;

  cinv cframe(D[x])
  call Proc4();

  assert D[x] = 5;
```

Fig. 3. Example with dynamic memory allocation and the Inverse set constructor.

The resultant annotated program is verified by `Boogie` by generating a logical formula (verification condition) and checking the formula with *satisfiability modulo theories* (SMT) solvers. The assertion logics used in these programs use sophisticated set constructors in addition to the usual theories of uninterpreted functions, select-update arrays and arithmetic supported by most SMT solvers. In spite of the complexity, the assertion logics used in these examples are closed under the *wp* predicate transformer with respect to call-free and loop-free statements in the language. For a few cases, we even have decision procedures (i.e. a sound, complete and terminating procedure) for deciding formulas in the assertion logic [6, 12].

4.1 Dynamic allocation

In the example in Figure 3, we consider a map Alloc whose range contains two Boolean values true and false. The map is used to track the set of *allocated* elements of the domain; $\mathsf{Alloc}[u] = \mathsf{true}$ if and only if u is an allocated element. The statement x := new is a sugar for the following statements: {var u in x := u endvar; **assume** Alloc[x] = false; Alloc[x] := true; }. In this example, the map D is mutated at an unbounded set of freshly allocated locations in procedure *Proc4* — namely at the locations u for which $\mathsf{Alloc}[u] = \mathsf{false}$ at entry to *Proc4*. In this case, this modified set excludes x in *Proc3*.

To specify the modified set, we use the set constructor $\mathsf{Inverse} : (\mathsf{int} \to \mathsf{int}) * \mathsf{int} \to 2^{\mathsf{int}}$, which takes a map and returns all elements of the domain that map to a given value; i.e. $\mathsf{Inverse}(\mathsf{X}, v) \doteq \{u \mid \mathsf{X}[u] = v\}$. For this set constructor, $wp(\mathsf{X}[\mathsf{x}] := \mathsf{y}, u \in \mathsf{Inverse}(\mathsf{X}, v))$ is given by [12]:

$$(\mathsf{y} = v \wedge u \in \mathsf{Inverse}(\mathsf{X}, v) \cup \{\mathsf{x}\}) \vee (\mathsf{y} \neq v \wedge u \in \mathsf{Inverse}(\mathsf{X}, v) \setminus \{\mathsf{x}\})$$

4.2 Linked lists

Reverse Consider the recursive implementation of list reversal in Figure 4(a). This implementation performs an in-place reversal of an input list. The precondition requires the argument h to point to a nonempty acyclic list. The procedure

```
var N : int → int;

pre h ≠ nil
pre nil ∈ Btwn(N, h, nil)
updates N @ Btwn(N, h, nil)
post Btwn(old(N), h, nil) = Btwn(N, r, nil)
post ∀u ∈ Btwn(old(N), h, nil).
        u = nil ∨
        Btwn(old(N), h, u) = Btwn(N, u, h)
proc reverse(h : int) : (r : int) =
  if (N[h] = nil) {
    r := h;
  } else {
    cinv cframe(N[h])
    r := reverse(N[h]);
    N[N[h]] := h;
    N[h] := nil;
  }
```

```
var N : int → int;
var D : int → int;


pre r ∈ Btwn(N, l, r)
updates N @ ROS(N, l, r);
post Sorted(N, D, hd, r)
post r ∈ Btwn(N, hd, r)
post ROS(old(N), l, r) = ROS(N, hd, r)
proc quick_sort(l : int, r : int) returns (hd : int) =
  var ret : int;

  if (l = r ∨ N[l] = r) {
    hd := l;
  } else {
    hd := partition(l, r);
    cinv cframe(ROS(N, hd, N[l]));
    cinv cframe(N[l]);
    ret := quick_sort(N[l], r);
    N[l] := ret;
    cinv cframe(ROS(N, N[l], r));
    cinv Sorted(N, D, N[l], r);
    ret := quick_sort(hd, N[l]);
    hd := ret;
}
```

Fig. 4. List examples (a) reverse, (b)list-based quick sort

may modify the N map only at the pointers in this list. The first postcondition asserts that the set of elements in the output list is the same as the set of elements in the input list. The second postcondition strengthens this assertion to ensure that the ordering in the output list is the reverse of the ordering in the input list.

The recursive call to *reverse* requires a call invariant stating that the value of N[h] remains unchanged by the call. This assertion is crucial for ensuring that subsequent updates to N first, do not trash the list reversal performed by the recursive call itself and second, successfully reverse the link from h to N[h].

List sort We have also verified an implementation of quick sort for lists. This example (present in Figure 4(b)), required nontrivial call invariants. We have used the following helper predicates to define the annotations for this example:

$$
\begin{aligned}
&ROS(X, u, v) && \doteq \mathsf{Btwn}(X, u, v) \setminus \{v\} \\
&UpperBound(X, Y, l, r, d) && \doteq \forall u \in ROS(X, l, r) : Y[u] \leq d \\
&LowerBound(X, Y, l, r, d) && \doteq \forall u \in ROS(X, l, r) : Y[u] \geq d \\
&Sorted(X, Y, l, r) && \doteq \forall u \in ROS(X, l, r) : \forall v \in ROS(X, u, r) : Y[u] \leq Y[v]
\end{aligned}
$$

```
var A : int → int;

post idx ∈ [l, r)
post Deref(A, [l, r)) = Deref(old(A), [l, r))
updates A @ [l, r)
proc Partition(l : int, r : int, pivot : int) : (idx : int) =
  /* Partitions A and returns the final index of pivot */

pre 0 < l ≤ r
post Deref(A, [l, r)) = Deref(old(A), [l, r))
updates A @ [l, r)
proc QuickSort(l : int, r : int) : void =
  var pivot : int, idx : int;
  if (l = r) return;
  pivot := A[l];
  idx := Partition(l, r, pivot);
  cinv cframe(Deref(A, [idx, r)));
  QuickSort(l, idx);
  cinv cframe(Deref(A, [l, idx + 1)));
  QuickSort(idx + 1, r);
```

Fig. 5. Example of quicksort over an array A.

The example requires reasoning about shapes of lists, properties of a collection of pointers, and arithmetic relationship on the data elements of the list. The recursive nature of the procedure makes the proof highly non-trivial — which explains the complexity of the call invariants. The proof illustrates the benefits of combination frameworks present in first-order theorem provers for precise reasoning of such examples.

Merge and append In addition to these programs, we have also successfully verified recursive implementations for appending and merging two lists. These implementations and their specifications are described in the appendix. Interestingly, in spite of the presence of recursive calls and unbounded number of updates, no call invariants were required for proving the correctness of these examples.

4.3 Arrays

In this example, we illustrate the use of two new set constructors, (i) the range set constructor $[i, j)$, and (ii) a set constructor Deref to collect the content of a set of locations. Consider the quicksort algorithm in Figure 5 where the array map A is being sorted with recursive invocations to *QuickSort*. The procedure *QuickSort* sorts the indices of the array A in the range $[\mathsf{l}, \mathsf{r})$. The procedure *Partition* (we omit the procedure body) takes a value pivot and returns an index idx such that $\mathsf{idx} \in [\mathsf{l}, \mathsf{r})$.

Let us check a simple property that the algorithm preserves the contents of A, assuming distinct elements in the array. Since the postcondition of *Partition* establishes this constraint, the main challenge is in establishing this fact across the two recursive calls to *QuickSort*. The two call invariants serve to preserve facts at the call-site required to establish the postcondition. For example, the first call invariant states that the contents of A in the range $[\mathsf{idx}, \mathsf{r})$ is preserved by the call to $\mathit{QuickSort}(\mathsf{l}, \mathsf{idx})$ since the procedure does not modify these locations.

The specifications for this example refer to a dependent set constructor $\mathsf{Deref} : (\mathsf{int} \to \mathsf{int}) * 2^{\mathsf{int}} \to 2^{\mathsf{int}}$ that takes a map and a set and constructs a set with the union of values of the map at elements in the set. Formally, $\mathsf{Deref}(\mathsf{X}, S) \doteq \{\mathsf{X}[u] \mid u \in S\}$. Interestingly, this assertion logic is still closed under *wp* with respect to statements of our language. For this set constructor, $\mathit{wp}(\mathsf{X}[\mathsf{x}] := \mathsf{y}, u \in \mathsf{Deref}(\mathsf{X}, S))$ is defined as follows:

$$\bigvee \mathsf{x} \notin S \wedge u \in \mathsf{Deref}(\mathsf{X}, S)$$
$$\bigvee \mathsf{x} \in S \wedge \mathsf{X}[\mathsf{x}] \in \mathsf{Deref}(\mathsf{X}, S \setminus \{\mathsf{x}\}) \wedge u \in \mathsf{Deref}(\mathsf{X}, S) \cup \{\mathsf{y}\}$$
$$\bigvee \mathsf{x} \in S \wedge \mathsf{X}[\mathsf{x}] \notin \mathsf{Deref}(\mathsf{X}, S \setminus \{\mathsf{x}\}) \wedge u \in (\mathsf{Deref}(\mathsf{X}, S) \setminus \{\mathsf{X}[\mathsf{x}]\}) \cup \{\mathsf{y}\}$$

The first disjunct corresponds to the case when $\mathsf{Deref}(\mathsf{X}, S)$ remains unchanged; the second (and the third) disjunct corresponds to the cases when the value $\mathsf{X}[\mathsf{x}]$ is contained (respectively not contained) in X at an index other than x.

4.4 Discussion

We now discuss the call invariant annotations highlighted as **cinv .** . These call invariants are interesting because their specification can be obviated by adding the following postcondition automatically for each procedure:

$$\mathbf{post}\ \forall u : \mathsf{int} :: u \in \mathsf{MS}_{\mathsf{X}} \vee \mathsf{X}[u] = \mathbf{old}(\mathsf{X})[u]$$

This postcondition ensures that the map X is preserved at a location disjoint from MS_{X}. However, the postcondition introduces a quantifier, which may compromise the termination of a theorem prover. For example, the decision procedure for reasoning about linked lists with sets [12] may not terminate when reasoning with quantified facts of the above form. The invariants marked with **cinv .** are the ones that do not have to be specified, when the theorem prover can prove the program with this quantified postcondition. The reader may observe that these are precisely those call invariants that do not contain a set constructor or a predicate that depends on the map being modified by the callee. This explains why most call invariants using Btwn are not removed.

The example of the list implementation of quick sort uses a call invariant which is not specified using the $\mathsf{cframe}(.)$ syntactic sugar. We needed to specify a single-state predicate to specify the sortedness of the list, instead of a set or a scalar expression inside $\mathsf{cframe}(.)$ — one may view this as a way to preserve a relation (in this case sortedness). Finally, several examples did not need a call invariant (e.g. list append and merge) even in the presence of unbounded updates in callees. This is due to the fact that these recursive procedures are actually tail-recursive, where one does not need to carry facts before the procedure call.

5 Related work

In this work, we provide a simple approach for leveraging precise verifiers for call-free programs to reason about programs with procedure calls. An important benefit of our approach is that it does not require adding additional unchecked or complex frame axioms [13] to the verification conditions. However, the benefit comes at the cost of additional user-specified annotations. Our contribution is complementary to the large body of work on modular verification in the presence of data hiding [5, 11, 2]. The implementations of these approaches invariably use complex quantifiers to encode variants of frame axioms that make verification unpredictable [1]; our work could potentially be used to eliminate or simplify these frame axioms.

Separation logic [17] is a specialization of Floyd-Hoare logic that requires a specialized assertion logic. The assertion logic contains formulas to describe heaps, where the formula $\phi * \psi$ denotes a heap with two disjoint subheaps for which ϕ and ψ hold respectively. The following *frame rule* [15] allows preservation of the fact R whose domain does not intersect with the modified locations in the statement s.

$$\text{[Frame rule]} \quad \frac{\{P\}\ s\ \{Q\}}{\{P * R\}\ s\ \{Q * R\}}$$

However, the specialized extension prevents leveraging existing tools for doing precise verification of call-free programs. Many of the inference rules for Floyd-Hoare logic do not apply in a straightforward manner; for example, the rule of constancy is no longer sound [17] in this extension. In addition, the specifications of s has to precisely describe the locations in the heap that s reads or write from, which is not required in Floyd-Hoare logic. Besides, one cannot use the standard *wp*-based methods to generate verification conditions. Although scalable and automatic shape analysis engines have been recently developed based on separation logic [21], proof obligations for expressive separation logic properties are often checked with higher order theorem provers [14].

The frame problem due to procedure scoping has also been explored in the context of automatic shape analysis [18, 9], where the caller's heap is separated from the callee's heap by identifying a set of *cutpoints* which are dominators for any location in the heap of the callee. These cutpoints are treated as ghost parameters, but precision is lost when when the set of cutpoints can be unbounded. We believe that call invariants may alleviate the need to introduce cutpoints to pass a local heap to a callee.

6 Conclusion

This paper makes two important contributions. First, we provide an automatic program instrumentation to address the imprecision due to procedure scoping. The call invariant annotation allows a user to specify an inductive hypothesis while dealing with unbounded updates in a callee. Such an instrumented and annotated program can be verified using any off-the-shelf verifier without any

need to interact with the lower-level theorem prover to specify the inductive hypothesis. This has allowed us to leverage existing precise verifiers for call-free programs to verify non-trivial examples with a small annotation burden. Second, we have separated the problem of specifying the frame from inferring the frame automatically for procedure calls. We imagine exploiting various loop invariant inference algorithms to synthesize most call invariants, given their restricted shape in practice. However, it still allows the user to explicitly specify the frame when the inference algorithm fails to discover the necessary frame.

References

1. A. Banerjee, M. Barnett, and D. A. Naumann. Boogie meets regions: A verification experience report. In *Verified Software: Theories, Tools, Experiments (VSTTE '08)*, LNCS 5295, pages 177–191, 2008.
2. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *Object-Oriented Programming, 22nd European Conference (ECOOP'08)*, LNCS 5142, pages 387–411. Springer, 2008.
3. M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO '05)*, LNCS 4111, pages 364–387, 2005.
4. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE '05)*, pages 82–87, 2005.
5. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, LNCS 3362, pages 49–69, 2005.
6. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation (VMCAI '06)*, LNCS 3855, pages 427–442, 2006.
7. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.
9. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis Symposium (SAS '06)*, LNCS 4134, pages 240–260. Springer, 2006.
10. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
11. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Symposium on Formal Methods (FM '06)*, LNCS 4085, pages 268–283, 2006.
12. S. K. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification using SMT Solvers. In *Principles of Programming Languages (POPL '08)*, pages 171–182, 2008.
13. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst.*, 24(5):491–553, 2002.

14. A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Programming Language Design and Implementation (PLDI '07)*, pages 468–479, 2007.
15. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL 01: 10th International Workshop on Computer Science Logic*, LNCS 2142, pages 1–19. Springer, 2001.
16. Z. Rakamarić, J. Bingham, and A. J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '06)*, LNCS 4349, pages 106–121, 2007.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS '02)*, pages 55–74, 2002.
18. N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Symposium on Principles of Programming Languages (POPL '05)*, pages 296–309. ACM, 2005.
19. Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.
20. Windows Driver Kit. http://www.microsoft.com/whdc/devtools/wdk/default.mspx.
21. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *Computer Aided Verification (CAV '08)*, LNCS 5123, pages 385–398, 2008.

A Weakest liberal precondition for simple statements

Figure 6 shows the weakest precondition for the simple statements in our language, and are fairly standard [4]. We use $\phi[e/\mathsf{x}]$ to denote syntactic substitution of all occurrences of a variable x with e in the formula ϕ.

$$
\begin{array}{ll}
wp(\mathsf{skip}, \phi) & = \phi \\
wp(\textbf{assert } \psi, \phi) & = \psi \wedge \phi \\
wp(\textbf{assume } \psi, \phi) & = \psi \implies \phi \\
wp(\mathsf{x} := e, \phi) & = \phi[e/\mathsf{x}] \\
wp(\mathsf{X} := \mathsf{Y}, \phi) & = \phi[\mathsf{Y}/\mathsf{X}] \\
wp(\mathsf{X}[e] := \mathsf{y}, \phi) & = \phi[\mathsf{X}[e := \mathsf{y}]/\mathsf{X}] \\
wp(\mathsf{var\ x\ in}\ s\ \mathsf{endvar}, \phi) & = \forall x : wp(s, \phi) \\
wp(\mathsf{var\ X\ in}\ s\ \mathsf{endvar}, \phi) & = \forall X : wp(s, \phi) \\
wp(s; t, \phi) & = wp(s, wp(t, \phi)) \\
wp(\mathsf{if}\ (e)\ \mathsf{then}\ s\ \mathsf{else}\ t, \phi) & = (e \neq 0 \wedge wp(s, \phi)) \vee (e = 0 \wedge wp(t, \phi))
\end{array}
$$

Fig. 6. Weakest precondition for simple statements (without loops and procedure calls).

For any $\phi \in \mathit{Formula}$, it is easy to see the $wp(s, \phi) \in \mathit{Formula}$ for most commmands s. For the statement $\mathsf{X}[e] := \mathsf{y}$, the resultant formula contains an update of a map, which is captured by the select-update theory of arrays.

B Multiple maps

Here is the program instrumentation in the presence of multiple maps being modified by a procedure call. Let us consider the case of two maps X and Y — the generalization is evident from the case of two maps.

$$
\begin{array}{l}
\mathsf{var}\ \mathsf{X}_{\mathsf{pre}}, \mathsf{X}_{\mathsf{post}}, \mathsf{Y}_{\mathsf{pre}}, \mathsf{Y}_{\mathsf{post}}\ \mathsf{in} \\
\quad (\mathsf{X}_{\mathsf{pre}}, \mathsf{Y}_{\mathsf{pre}}) := (\mathsf{X}, \mathsf{Y}); \\
\quad \mathsf{call}\ Foo(e); //\text{procedure call} \\
\quad (\mathsf{X}_{\mathsf{post}}, \mathsf{Y}_{\mathsf{post}}) := (\mathsf{X}, \mathsf{Y}); \\
\quad (\mathsf{X}, \mathsf{Y}) := (\mathsf{X}_{\mathsf{pre}}, \mathsf{Y}_{\mathsf{pre}}); \\
\\
\quad \textbf{inv}\ R[\mathsf{X}_{\mathsf{pre}}/\textbf{old}(\mathsf{X})][\mathsf{Y}_{\mathsf{pre}}/\textbf{old}(\mathsf{Y})] \\
\quad \mathsf{while}\ (*)\ \mathsf{do} \\
\quad\quad \mathsf{if}\ (*)\ \mathsf{var}\ \mathsf{u}, \mathsf{v}\ \mathsf{in}\ \textbf{assume}\ \mathsf{u} \in \mathsf{MS}_{\mathsf{X}}; \mathsf{X}[\mathsf{u}] := \mathsf{v};\ \mathsf{endvar} \\
\quad\quad \mathsf{if}\ (*)\ \mathsf{var}\ \mathsf{u}, \mathsf{v}\ \mathsf{in}\ \textbf{assume}\ \mathsf{u} \in \mathsf{MS}_{\mathsf{Y}}; \mathsf{Y}[\mathsf{u}] := \mathsf{v};\ \mathsf{endvar} \\
\\
\quad \textbf{assume}\ \mathsf{X} = \mathsf{X}_{\mathsf{post}}; \\
\quad \textbf{assume}\ \mathsf{Y} = \mathsf{Y}_{\mathsf{post}}; \\
\mathsf{endvar}
\end{array}
$$

C Examples

We provide the remaining examples in the appendix. We omit the formal specifications for these examples, and instead informally describe the specifications. Note that for the proof of both these examples, we did not require any call invariants.

C.1 Recursive list append

The `append` procedure takes two acyclic and disjoint lists starting at `p` and `q` respectively and appends the second list at the tail of the first list. We have shown that the list returned is an actual append of the two lists and the relative order of the nodes in each list remains unchanged. Finally the procedure modifies the map `Next` only at the pointers of list from `p`.

C.2 Recursive list merge

The procedure `merge` takes as input two acyclic and disjoint lists which are sorted according to the values in `Data`, and outputs a sorted list which contains the union of the two lists. The procedure only modifies the pointers in the union of the two lists.

```
var Next: [int]int;

procedure append(p: int, q: int) returns (r: int)
{
  var t: int;
  if (p == 0) {
    r := q;
  } else {
    call t := append(Next[p], q);
    Next[p] := t;
    r := p;
  }
}
```

Fig. 7. Recursive list append

```
var Next: [int]int;
var Data: [int]int;

procedure merge(p:int, q:int) returns (r: int)
{
  var a: int;
  if (p == 0) {
    r := q;
  } else if (q == 0) {
    r := p;
  } else if (Data[p] < Data[q]) {
    call a := merge(Next[p], q);
    Next[p] := a;
    r := p;
  } else {
    call a := merge(p, Next[q]);
    Next[q] := a;
    r := q;
  }
}
```

Fig. 8. Recursive list merge