Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2007-049 October 31, 2007

Set Interfaces for Generalized Typestate and Data Structure Consistency Verification

Patrick Lam, Karen Zee, Viktor Kuncak, and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Set Interfaces for Generalized Typestate and Data Structure Consistency Verification

Patrick Lam, Karen Zee, Viktor Kuncak, and Martin Rinard

MIT Computer Science and Artificial Intelligence Laboratory, USA
{plam,kkz,vkuncak,rinard}@csail.mit.edu

Abstract

Typestate systems allow the type of an object to change during its lifetime in the computation. Unlike standard type systems, they can enforce safety properties that depend on changing object states. We present a new, generalized formulation of typestate that models the typestate of an object through membership in abstract sets. This abstract set formulation enables developers to reason about cardinalities of sets, and in particular to state and verify the condition that certain sets are empty. We support hierarchical typestate classifications by specifying subset and disjointness properties over the typestate sets.

We present our formulation of typestate in the context of the Hob program specification and verification framework. The Hob framework allows the combination of typestate analysis with powerful independently developed analyses such as shape analyses or theorem proving techniques. We implemented our analysis and annotated several programs (75-2500 lines of code) with set specifications. Our implementation includes several optimizations that improve the scalability of the analysis and a novel loop invariant inference algorithm that eliminates the need to specify loop invariants. We present experimental data demonstrating the effectiveness of our techniques.

1 Introduction

Typestate systems [7, 10, 12, 13, 21, 37] allow the type of an object to change during its lifetime in the computation. Unlike standard type systems, typestate systems can enforce safety properties that depend on changing object states.

[1] This research was supported by DARPA Contract F33615-00-C-1692, NSF Grant CCR-0086154, NSF Grant CCR-0073513, NSF Grant CCR-0209075, and the Singapore-MIT Alliance.

[2] This is a revised version of the paper [26]. The present version contains a new technique for loop invariant inference, and improves the presentation of the system.

This report was written in September 2006.

This paper develops a new, generalized formulation of typestate systems. Instead of associating a single typestate with each object, our system models each typestate as an abstract set of objects. Objects may, of course, simultaneously belong to multiple sets. If an object is in a given typestate, it is a member of the set that corresponds to that typestate. This formulation immediately leads to several generalizations of the standard typestate approach. It is possible to relate typestate sets by specifying subset and disjointness properties over sets, which enables our approach to support hierarchical typestate classifications. Furthermore, the use of the boolean algebra of sets to reason about set membership enables our approach to reason about cardinalities of sets, and in particular to state and verify that certain sets are empty. Finally, a typestate in our formulation can be formally related to a potentially complex property of an object, with the relationship between the typestate and the property verified using powerful independently developed analyses such as shape analyses or theorem provers.

We implemented the idea of generalized typestate in the Hob program specification and verification framework [24–28]. This framework supports the division of the program into instantiable, separately analyzable modules. Modules encapsulate private state and export abstract sets of objects that support abstract reasoning about the encapsulated state. Abstraction functions specify the objects that participate in each abstract set, and are defined using unary predicates on the encapsulated state. Modules also export procedures that may access the encapsulated state and therefore change the contents of the exported abstract sets. Each module uses set algebra expressions involving operators such as set union or difference to specify the preconditions and postconditions of exported procedures. As a result, the analysis of client modules that coordinate the actions of other modules can reason solely in terms of the exported abstract sets and avoid the complexity of reasoning about any encapsulated state.

When the encapsulated state implements a data structure (such as a list, hash table, or tree), the resulting abstract sets characterize how objects participate in that data structure. The developer can then use the abstract sets to specify consistency properties that involve multiple data structures from different modules. Such a property might state, for example, that two data structures involve disjoint objects or that the objects in one data structure are a subset of the objects in another. In this way, our approach captures global sharing patterns and characterizes both local and global data structure consistency.

The verification of a program in our system consists of the application of potentially different specialized analyses to verify 1) the set interfaces of all of the modules in the program and 2) the validity of the global data structure consistency properties. The set specifications separate the analysis of a complex program into independent verification tasks, where each task is verified by an appropriate specialized *analysis plugin* [24]. Our approach therefore makes it possible, for the first time, to apply multiple specialized, extremely precise, and unscalable analyses such as shape analysis [31, 34] or even manually aided theorem proving [38] to effectively verify sophisticated typestate and data structure consistency properties in sizable programs.

Specification Language. Our specification language is the full first-order theory of the boolean algebra of sets. In addition to basic typestate properties expressible using quantifier-free boolean algebra expressions, our language can state constant bounds on the cardinalities of sets of objects, such as "a local variable is not null" or "the content of the queue is nonempty", or even "the data structure contains at least one and at most ten objects". Because a cardinality constraint counts all objects that satisfy a given property, our specification language goes beyond standard typestate approaches that use per-object finite state machines. Our specification language also supports quantification over sets. Universal set quantifiers are useful for stating parametric properties; existential set quantifiers are useful for information hiding. Note that quantification over sets is not directly expressible even in such sophisticated languages as first-order logic with transitive closure. Despite this expressive power, our set specification language is decidable and furthermore extends naturally to Boolean Algebra with Presburger Arithmetic [22, 23].

The Flag Analysis Plugin. The generalized typestate analysis in the Hob system is implemented in the *flag analysis plugin*, which is the focus of this paper. The flag analysis plugin uses the values of integer and boolean object fields (flags) to define the meaning of abstract sets. It verifies set specifications by first constructing set algebra formulas whose validity implies the validity of the set specifications, then verifying these formulas using an off-the-shelf decision procedure [19].

The flag analysis plugin is important for two reasons. First, flag field values often reflect the high-level conceptual state of the entity that an object represents, and flag changes correspond to changes in the conceptual state of the entity. By using flags in preconditions of object operations, the developer can specify key object state properties required for the correct processing of objects and the correct operation of the program. Unlike standard typestate approaches, our flag analysis plugin can enforce not only temporal operation sequencing constraints, but also the generalizations that our expressive set specification language enables.

Second, the flag analysis plugin can propagate constraints between abstract sets defined with arbitrarily sophisticated abstraction functions in external modules. The plugin can therefore analyze modules that, as they coordinate the operation of other modules, indirectly manipulate external data structures defined in those other modules. This enables the flag analysis to perform the intermodule reasoning required to verify global invariants relating different data structures, *e.g.* inclusion and disjointness of data structures. Because the flag plugin uses the boolean algebra of sets to internally represent its dataflow facts, it can propagate and verify these constraints in a precise way.

Evaluation. We implemented our flag analysis plugin in the context of the Hob system [27, 28]. In addition to the flag analysis plugin, the Hob system contains a shape analysis plugin based on Pointer Assertion Logic Tool [31], and a theorem proving plugin [38] that uses a verification-condition generator and the Isabelle interactive proof assistant [32]. We used the flag analysis plugin to verify high-level properties in our benchmarks; we used the other two plugins to verify implementations of encapsulated data structures. Overall, most of the code was verified using the scalable flag analysis plugin, allowing the more precise analyses to be focused on the intricacies of internal data structure implementations.

Our initial implementation of the flag analysis algorithm simply synthesized boolean algebra formulas and used the MONA decision procedure [19] directly to discharge them. We found that scalability problems with the MONA decision procedure prevented this initial approach from analyzing some of our benchmarks. We therefore implemented several formula simplifications that substantially improved the scalability of the flag analysis; we present experimental data that show the effect of our formula simplifications.

Loop invariant inference. Our flag analysis is based on symbolically computing the postconditions of statements, which makes it precise. A general problem with such an approach is the handling of loops. Previously, our analysis used a simple loop invariant inference technique that was often ineffective at deriving loop invariants; developers would typically be forced to supply loop invariants explicitly. Like procedure summaries, invariants provide useful information for code understanding; however, unlike procedure summaries, they are not essential for modular analysis. To eliminate the necessity of writing loop invariants, we have therefore developed a more sophisticated loop invariant inference technique, which we present in Section 6. We found that our loop invariant inference technique was successful in inferring all loop invariants in our benchmarks.

2 Example

In this section we illustrate how Hob analyzes a program consisting of multiple modules and explain the role of the flag analysis plugin in Hob.

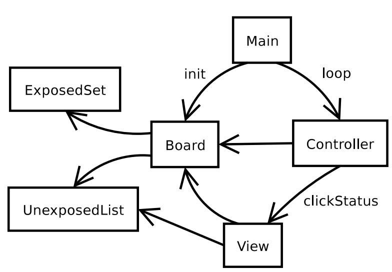

Fig. 1. Modules in Minesweeper implementation

We use an implementation of the popular Minesweeper game as an example.

```
impl module Board {
  format Cell {
    isMined : bool;
    isExposed : bool;
    isMarked : bool;
    i, j : int;
    init : bool;
  }
  var init, peeking, gameOver : bool;

  proc revealAllUnexposed() {
    UnexposedList.openIter();
    bool b = UnexposedList.isLastIter();
    while (!b) {
      Cell c = UnexposedList.nextIter();
      UnexposedList.remove(c);
      c.isExposed = true;
      ExposedSet.add(c);
      b = UnexposedList.isLastIter();
    }
  }
  ...
}
```

Fig. 2. Implementation of Board

```
spec module Board {
    format Cell;
    specvar MarkedCells, MinedCells,
            ExposedCells, UnexposedCells,
            U : Cell set;



    specvar init, peeking, gameOver : bool;

    proc revealAllUnexposed()
      requires gameOver & init & not peeking
      modifies ExposedCells, UnexposedCells
      ensures card(UnexposedCells') = 0;



    ...



}
```

Fig. 3. Specification of Board

```
abst module Board {
    use plugin "flags";
    U = { x : Cell | "x.init = true" };
    MarkedCells = U cap { x : Cell | "x.isMarked = true" };
    ExposedCells = U cap { x : Cell | "x.isExposed = true" };
    UnexposedCells = U cap { x : Cell | "x.isExposed = false" };
    MinedCells = U cap { x : Cell | "x.isMined = true" };
    predvar gameOver; predvar init; predvar peeking;
}
```

Fig. 4. Abstraction of Board

Figure 1 presents the module diagram of our minesweeper implementation, with boxes representing modules and arrows representing procedure calls.[3] Our minesweeper implementation has several modules: a game board module (which represents the game state), a controller module (which responds to user input), a view module (which produces the game's output), an exposed cell module (which stores the exposed cells in an array), and an unexposed cell module (which stores the unexposed cells in an instantiated linked list).

Each module in Hob consists of three sections: the implementation section, the specification section, and the abstraction section. Our minesweeper implementation uses the standard model-view-controller (MVC) design pattern [16]; the `Board` module implements the model part of the MVC pattern. Figures 2, 3, and 4 present the three sections of the `Board` module.

The implementation section contains the executable code for each procedure of the module, written in a type-safe imperative language similar to Java or ML. In this example we examine the `revealAllUnexposed` procedure, which

[3] Full source code for the minesweeper example and other case studies, the interpreter, the Java translator, and the Hob analysis engine are available at the Hob homepage, `http://hob.csail.mit.edu`. The Hob page is served by a custom web server implemented in the Hob language.

is called at the end of the game to reveal the positions of all cells that have not been exposed so far. In addition to procedure implementations, the implementation section contains declarations of private global variables, such as the boolean variables `gameOver`, `init`, and `peeking` in Figure 2, and field declarations, such as `isMined`, `isExposed`, and `isMarked`. These fields reflect the fact that each `Cell` object may represent a mined, exposed or marked cell in the minesweeper game. Field declarations are grouped into *formats*. Multiple modules can contribute fields to the same format, allowing encapsulation at the granularity of fields [25].

The specification section contains the public interface for the module, expressed in terms of specification variables, including the global set-valued variables `MarkedCells`, `MinedCells`, `ExposedCells`, and `UnexposedCells`, and the global boolean variables in Figure 3. The specification section allows the clients of the module to reason about module behavior without having access to the implementation of the module. The specification module describes the behavior of each procedure using procedure contracts written in terms of the specification variables. For example, the contract of procedure `revealAllUnexposed` indicates 1) that the procedure may only be called when the `gameOver` variable is `true`, 2) that the only relevant specification variables that are modified are `ExposedCells` and `UnexposedCells`, and 3) that the size of the set `UnexposedCells` at the end of procedure execution is zero, that is, the set is empty. Section 3 describes our specification language more detail.

Finally, the abstraction section of the module specifies the mapping between the implementation section and the abstraction section, by defining each specification variable in terms of implementation variables. For example, the abstraction section in Figure 4 defines the set `UnexposedCells` as the set of all allocated objects whose `init` field is `true` and whose `isExposed` field is `false`. The abstraction section also indicates the name of the analysis plugin used to analyze the module; the `Board` module uses the flag plugin. The defining formula of each specification variable is given in a language specific to the plugin used to analyze the module; see [38] for another example of specification variable definitions.

Our system uses the flag analysis plugin to verify that our implementation has the following properties (among others):

- Unless the game is over, the set of mined cells is disjoint from the set of exposed cells.
- The sets of exposed and unexposed cells are disjoint.
- The set of unexposed cells maintained in the `Board` module is identical to the set of unexposed cells maintained in the `UnexposedList` list.
- The set of exposed cells maintained in the `Board` module is identical to the set of exposed cells maintained in the `ExposedSet` array.
- At the end of the game, all cells are revealed; that is, the set of unexposed cells is empty.

```
spec module UnexposedList {
  format Cell;
  specvar Content, Iter : Cell set;
  invariant Iter in Content;

  proc remove(n : Cell)
  requires card(n)=1 & (n in Content)
  modifies Content, Iter
  ensures (Content' = Content - n) &
          (Iter' = Iter - n);

  proc openIter()
  requires card(Iter) = 0
  modifies Iter
  ensures (Iter' = Content);

  proc isLastIter() returns e:bool
  ensures not e' <=> (card(Iter')>= 1);
  ...
}
```

Fig. 5. Specification of the `UnexposedList` module

```
scope Model {
modules Board, ExposedSet, UnexposedList;
exports Board;
invariant
 (Board.ExposedCells = ExposedSet.Content) &
 (Board.UnexposedCells = UnexposedList.Content) &
 (Board.init => ExposedList.setInit) &
 (Board.peeking |
             (card(UnexposedList.Iter) = 0));
}
```

Fig. 6. Scope for Minesweeper Example

To illustrate our flag analysis, we discuss the analysis of the `revealAllUnexposed` procedure. The goal of the analysis is to show that the implementation in Figure 2 conforms to the specification in Figure 3 when the specification variables are defined as in Figure 4. The procedure `revealAllUnexposed` invokes operations in the `UnexposedList` and `ExposedSet` modules, which implement sets using linked lists and arrays respectively. Figure 5 shows a fragment of the specification of the `UnexposedList` module. Hob's separation of modules into implementation, specification and abstractions sections enables analyses to examine only the specifications of called modules. For example, the analysis of the `Board` module need not handle the complexity of analyzing the implementation of `UnexposedList`. It simply uses the specification of `remove` in Figure 5 to derive the effect of a call `UnexposedList.remove(c)` in Figure 2. The analysis of `revealAllUnexposed` starts with the full precondition of `revealAllUnexposed`. The full precondition includes the explicitly stated requires clause in Figure 3, as well as the *scope invariant* in Figure 6. A *scope* in the Hob system is a collection of modules, some of which are exported, along with a list of scope invariants [25]. Scope invariants are global invariants that span specification variables from multiple modules; these invariants are implicitly conjoined to the preconditions and postconditions of all public procedures declared in exported modules, including the `revealAllUnexposed` procedure that we are discussing.

Starting from the precondition, the flag analysis uses a postcondition semantics of statements to compute an approximation of the transition relation between 1) the initial state of the procedure and 2) the state of the procedure at each program point. The flag analysis represents this approximation as a formula relating unprimed variables and primed variables. Upon entry to the procedure, the relation contains the precondition, as well as the conjuncts such as `UnexposedList.Iter'` = `UnexposedList.Iter` for each variable relevant to the analysis of the procedure, indicating that none of the variables have changed. When analyzing the first statement, `UnexposedList.openIter()`, the flag analysis checks that the current state implies the precondition of `openIter`, which follows from the clause `not peeking` from the `revealAllUnexposed` precondition, combined with the scope invariant. The analysis then uses the specification of `openIter` to derive a new transition relation formula that implies `Iter' = Content` (and does not contain the conjunct `UnexposedList.Iter' = UnexposedList.Iter`). Subsequent analysis derives properties that involve local variables; for instance, `b <=> card(UnexposedList.Iter')>=1` holds after a call to `UnexposedList.isLastIter`. The analysis of the loop proceeds by iterating the loop several times and removing the conjuncts that do not persist across all loop iterations. Section 6 describes our loop invariant inference algorithm in greater detail. Eventually the fixpoint iteration terminates and the analysis verifies that the synthesized loop invariant implies the postcondition, which consists of 1) the `ensures` clause and 2) the scope invariant.

Note that, in the course of its operation, our flag analysis verifies that the invoked procedures in `UnexposedList` are always used correctly. This usage constraint includes data structure operation preconditions: any element inserted into the list with `ExposedList.add(c)` must not already be in the list. Furthermore, our flag analysis propagates boolean conditions reflecting global game state information, such as `init`, `not peeking` and `gameOver`.[4] In the rest of this paper we describe the flag analysis of our framework in more detail.

3 Specification Language

Figure 7 presents the syntax for the specification section of modules in our language. This section contains a list of set definitions and procedure specifications and lists the names of types used in these set definitions and procedure specifications. Set declarations identify the module's abstract sets, while boolean variable declarations identify the module's abstract boolean variables. Each procedure specification contains a `requires`, `modifies`, and `ensures` clause. The `requires` clause identifies the precondition that the procedure requires to execute correctly; the `ensures` clauses identifies the postcondition that the procedure ensures when called in program states that satisfy the `requires` condition. The `modifies` clause identifies sets whose elements may change as a result of executing the procedure. For the purposes of this paper, `modifies` clauses can be viewed as a special syntax for a frame-condition conjunct in the `ensures` clause. The variables in the `ensures` clause can refer to both the initial (unprimed variables) and final (primed variables) states of the procedure. Both `requires` and `ensures` clauses use arbitrary first-order boolean algebra formulas B extended with cardinality constraints. A free vari-

[4] The Hob framework supports an additional *default* construct that allows the developer to specify conjuncts such as `init` and `not peeking` as default values that apply to a set of procedures given by some crosscut expression, so these conjuncts need not be repeated for every procedure [25].

$$
\begin{array}{rcl}
M & ::= & \textsf{spec module } m\ \{(\textsf{type } t)^*(\textsf{set } S)^*(\textsf{predvar } b)^* P^*\} \\
P & ::= & \textsf{proc } pn(p_1 : t_1, \ldots, p_n : t_n)[\textsf{returns } r : t] \\
 & & [\textsf{requires } B]\ \ [\textsf{modifies } S^*]\ \ \textsf{ensures } B \\
B & ::= & SE_1 = SE_2 \mid SE_1 \subseteq SE_2 \mid \textsf{card}(SE){=}k \\
 & \mid & B \wedge B \mid B \vee B \mid \neg B \mid \exists S.B \mid \forall S.B \\
SE & ::= & \emptyset \mid p \mid [m.]\,S \mid [m.]\,S' \\
 & \mid & SE_1 \cup SE_2 \mid SE_1 \cap SE_2 \mid SE_1 \setminus SE_2
\end{array}
$$

Fig. 7. Syntax of the Module Specification Language

$$
\begin{array}{rcl}
M & ::= & \textsf{abst module } m\ \{D^*\ P^*\} \\
D & ::= & \textsf{id}{=}D_r; \\
D_r & ::= & D_r \cup D_r \mid D_r \cap D_r \mid \textsf{id} \mid \{x : T \mid x.f{=}c\} \\
P & ::= & \textsf{predvar } p;
\end{array}
$$

Fig. 8. Syntax of the Flag Abstraction Language

able of any formula appearing in a module specification denotes an abstract set or boolean variable declared in that specification; it is an error if no such set or boolean variable has been declared. The expressive power of such formulas is the first-order theory of boolean algebras, which is decidable [20,30]. The decidability of the specification language ensures that analysis plugins can precisely propagate the specified relations between the abstract sets.

4 Overview of Flag Analysis

Our flag analysis verifies that modules implement set specifications in which integer or boolean flags indicate abstract set membership. The developer specifies (using the flag abstraction language) the correspondence between concrete flag values and abstract sets from the specification, as well as the correspondence between the concrete and the abstract boolean variables. Figure 8 presents the syntax for our flag abstraction modules. This abstraction language defines abstract sets in two ways: (1) directly, by stating a base set; or (2) indirectly, as a set-algebraic combination of sets. *Base sets* have the form $B = \{x : T \mid x.f{=}c\}$ and include precisely the objects of type T whose field `f` has value `c`, where `c` is an integer or boolean constant; the analysis converts mutations of the field `f` into set-algebraic modifications of the set B. *Derived sets* are defined as set algebraic combinations of other sets; the flag analysis handles derived sets by conjoining the definitions of derived sets (in terms of base sets) to each verification condition and tracking the contents of the base sets. Derived sets may use named base sets in their definitions; additionally, they may use *anonymous* sets given by set comprehensions. In that case, the flag analysis assigns internal names to anonymous sets and tracks their values to compute the values of derived sets.

Operation of the Analysis Algorithm. The flag analysis verifies a module M by verifying each procedure of M. To verify a procedure, the analysis performs abstract interpretation [5] with analysis domain elements represented by formulas. Our analysis associates quantified set algebra formulas B to each program point. A formula B has two collections of set variables: unprimed set variables S denoting initial values of sets at the entry point of the procedure, and primed set variables S' denoting the values of these sets at the current program point. B may also contain unprimed and primed boolean variables b and b' representing the pre- and post-values of local and global boolean variables. The definitions in the abstraction sections of the module provide the interpretations of these variables. The use of primed and unprimed variables allows our analysis to represent, for each program point p, a binary relation on states that overapproximates the reachability relation between procedure entry and p [6, 17, 35].

In addition to the abstract sets from the specification, the analysis also generates a set for each (object-typed) local variable. This set is either empty, indicating a null reference, or has cardinality one and contains the object to which the local variable refers. The formulas that the analysis manipulates therefore support the disambiguation of local variable and object field accesses at the granularity of the sets in the analysis; other analyses often rely on a separate pointer analysis to provide this information.

The initial dataflow fact at the start of a procedure is the precondition for that procedure, transformed into a relation by conjoining $S' = S$ for all relevant sets. At merge points, the analysis uses disjunction to combine set algebra formulas. The analysis allows the developer to provide loop invariants directly. If an invariant is not supplied, the analysis infers it using the algorithm in Section 6. After running the dataflow analysis, our analysis checks that the procedure conforms to its specification by checking that the derived postcondition (which includes the `ensures` clause and any required invariants and defaults [25]) holds at all exit points of the procedure. In particular, the flag analysis checks that for each exit point e, the computed formula B_e implies the procedure's postcondition.

Computing Postconditions. The transfer functions in the dataflow analysis update set algebra formulas to reflect the effect of each statement. Recall that the dataflow facts for the flag analysis are set algebra formulas B denoting a relation between the state at procedure entry and the state at the current program point. Let B_s be the set algebra formula describing the effect of statement s. The postcondition $B \circ B_s$ is the result of symbolically composing the relations defined by the formulas B and B_s. Conceptually, postcondition computation updates B with the effect of B_s. We compute $B \circ B_s$ by applying equivalence-preserving simplifications to the formula

$$\exists \hat{S}_1, \ldots, \hat{S}_n.\ B[S'_i \mapsto \hat{S}_i] \wedge B_s[S_i \mapsto \hat{S}_i]$$

Our flag analysis handles each statement in the implementation language by providing appropriate transfer functions for these statements. The generic transfer function is a relation of the form $[\![\mathrm{st}]\!](B) = B \circ \mathcal{F}(\mathrm{st})$, where $\mathcal{F}(\mathrm{st})$ is the formula symbolically representing the transition relation for the statement st expressed in terms of abstract sets. The transition relations for the statements in our implementation language are in Appendix A.

Verifying Implication of Dataflow Facts. A compositional program analysis needs to verify implication of constraints as part of its operation. Our flag analysis verifies implication when it encounters an assertion, procedure call, or procedure postcondition. In these situations, the analysis generates a formula of the form $B \Rightarrow A$ where B is the current dataflow fact and A is the claim to be verified. [5] The implication to be verified, $B \Rightarrow A$, is a formula in the boolean algebra of sets. We use the MONA decision procedure to check its validity [18], along with the transformations described in Section 5.

5 Boolean Algebra Formula Transformations

In our experience, applying several formula transformations drastically reduced the size of the formulas emitted by the flag analysis, as well as the time needed to determine their validity using an external decision procedure; in fact, some benchmarks could only be verified with the formula transformations enabled. This section describes the transformations we found to be useful. Section 8 presents our measurements of the improvements obtained from applying these transformations.

Smart Constructors. The constructors for creating boolean algebra formulas apply peephole transformations as they create the formulas. Constant folding is the simplest peephole transformation: for instance, attempting to create $B \land \mathsf{true}$ gives the formula B. Our constructors fold constants in implications, conjunctions, disjunctions, and negations. Similarly, attempting to quantify over unused variables causes the quantifier to be dropped: $\exists x.F$ is created as just F when x does not occur free within F. Most interestingly, we factor common conjuncts out of disjunctions: $(A \land B) \lor (A \land C)$ is represented as $A \land (B \lor C)$. Conjunct factoring greatly reduces the size of formulas tracked after control-flow merges, since most conjuncts are shared on both control-flow branches. The effects of this transformations appear similar to the effects of SSA form conversion in weakest precondition computation [15, 29].

Basic Quantifier Elimination. We symbolically compute the composition of statement relations while computing postconditions by existentially quantifying over all state variables. However, most relations corresponding to statements modify only a small part of the state and contain the frame condition that indicates that the rest of the state is preserved. The result of relation composition can therefore often be written in the form $\exists x.x = x_1 \land F(x)$, which is equivalent to $F(x_1)$. In this way we reduce both the number of conjuncts and the number of quantifiers. Moreover, this transformation can reduce some conjuncts to the form $t = t$ for some Boolean algebra term t, which is a true conjunct that is eliminated by further simplifications.

[5] Note that B may be unsatisfiable; this often indicates a problem with the program's specification. The flag analysis can, optionally, check whether B is unsatisfiable and emit a warning if it is. This check enabled us to improve the quality of our specifications by identifying errors in specifications.

It is instructive to compare our technique to weakest precondition computation [15] and forward symbolic execution [4]. These techniques are optimized for the common case of assignment statements and perform relation composition and quantifier elimination in one step. Our technique achieves the same result, but is methodologically simpler and applies more generally. In particular, our technique can take advantage of equalities in transfer functions that are not a result of analyzing assignment statements, but are given by explicit formulas in `ensures` clauses of procedure specifications. Such transfer functions may specify more general equalities such as $A = A' \cup x \wedge B' = B \cup x$ which do not reduce to simple backward or forward substitution.

Leveraging Quantifier Elimination in Implications We rewrite $\forall x.f \Rightarrow g$ as $\neg(\exists x.f \wedge \neg g)$. This greatly increases the applicability of the quantifier-elimination optimization described above.

Quantifier Nesting. We have experimentally observed that the MONA decision procedure works substantially faster when each quantifier is applied to the smallest scope possible. We have therefore implemented a quantifier nesting step that reduces the scope of each quantifier to the smallest possible subformula that contains all free variables in the scope of the quantifier. For example, our transformation replaces the formula $\forall x.\ \forall y.\ (f(x) \Rightarrow g(y))$ with $(\exists x.\ f(x)) \Rightarrow (\forall y.\ g(y))$.

To take maximal advantage of our transformations, we simplify formulas after relation composition and before invoking the decision procedure. Our global simplification step rebuilds formulas bottom-up and applies simplifications to each subformula.

6 Loop Invariant Synthesis

In this section, we summarize how our flag analysis plugin handles loops. The plugin can either verify developer-provided loop invariants or synthesize loop invariants from the program source code and specifications.

Explicit Loop Invariants. If the developer provides an explicit loop invariant, the plugin verifies that the loop invariant: 1) holds on entry to the loop; and 2) is preserved by the loop body. At the exit of the loop, the loop invariant conjoined with the loop exit condition characterizes the post-loop program state.

Our loop invariant verification algorithm uses information from the loop's context to automatically augment the explicit loop invariant with properties that are known to be invariant over the loop. In particular, the loop's containing procedure has a `requires` clause, which states the procedure precondition. This clause involves only the initial values of sets at the begining of the procedure (unprimed set variables), and therefore holds throughout the procedure execution, including within the loop body. We also use the containing pro-

```
Infer-Loop-Invariant(f0, loop-condition, loop-body, max-iterations)
  i ← 0
  f ← f0
  f′ ← Compute-Postcondition(f ∧ loop-condition, loop-body)
  while i < max-iterations and f′ ⇏ f
     do f ← Get-Implied-Conjuncts(f, f′, []) ∧ Get-Implied-Conjuncts(f′, f, [])
        f′ ← Compute-Postcondition(f ∧ loop-condition, loop-body)
        i ← i + 1
  if i ≥ max-iterations
     then while f′ ⇏ f
             do f ← Get-Implied-Conjuncts(f, f′, [])
                f′ ← Compute-Postcondition(f ∧ loop-condition, loop-body)
  return f

Get-Implied-Conjuncts(f1, f2, [x0, ..., xn])
  result ← True
  foreach c in Conjuncts(f1)
     if f2 ⇒ ∃x0, ..., xn.c
        then result ← c ∧ result
     else if c has the form ∃x.e
        then result ← Handle-Existential(e, f2, [x0, ..., xn, x]) ∧ result
  return result

Handle-Existential(e, f, [x0, ..., xn])
  g ← Get-Implied-Conjuncts(e, f, [x0, ..., xn])
  if f ⇒ ∃x0, ..., xn.g
     then return ∃xn.g
  g ← True
  foreach c in Conjuncts(e)
     if c does not contain xn
        then g ← c ∧ g
  return Get-Implied-Conjuncts(g, f, [x0, ..., x(n−1)])
```

Fig. 9. Pseudo-code for Loop Invariant Inference Algorithm.

cedure's implementation, as well as its `modifies` clause, to identify all non-modified sets, and construct a conjunct which states that these non-modified sets are preserved by the loop[6]. We then conjoin both the original procedure precondition and clauses guaranteeing the preservation of non-modified sets to all explicit loop invariants. Developers therefore need not provide these two pieces of redundant information, which helps to make explicit invariants more concise and easier to understand.

Inferred Loop Invariants. If the developer does not provide an explicit loop invariant, the flag analysis automatically synthesizes one. The synthesis starts with the formula characterizing the transition relation at the entry of the procedure and weakens the formula by iterating the analysis of the loop until it reaches a fixpoint. Figure 9 presents pseudocode for the algorithm. In the remainder of this section we present an example of the algorithm in action, discuss some properties of the algorithm, and present our experience with the algorithm applied to our set of benchmarks.

Example. Figure 10 presents procedure `clear`, which iterates through a set,

[6] Using the procedure's `modifies` clause alone results in an overly-conservative estimate of modified *private* sets in the presence of scopes [25], because scope-public procedures do not declare modifications of scope-private sets. Our use of the `modifies` clause, on the other hand, allows the developer to state more detailed information about *public* sets than our modified-set inference algorithm could deduce.

```
specvar Content : Element set;

proc clear()    // specification
  requires true
  modifies Content
  ensures card(Content') = 0;

proc clear() {  // implementation
    pre:  bool e; e = isEmpty();
    head: while (!e) {
    body:     Entry q = removeFirst();
              e = isEmpty();
          }
    post: return;
}
```

Fig. 10. Procedure containing a loop.

```
proc isEmpty() returns b : bool
  ensures not b' <=> card(Content)>=1

proc removeFirst() returns e : Element
  requires card(Content)>0
  modifies Content
  ensures (card(e')=1) & (e' in Content) &
          (Content' = Content - e');
```

Fig. 11. Procedures called within the loop.

removing each element until the set is empty. We use this procedure to illustrate our loop inference technique. In procedure `clear`, each execution of the loop body removes an element from the `Content` set. Because the precondition of procedure `removeFirst` must hold, the loop body cannot execute successfully unless the `Content` set is non-empty, i.e. `card(Content') >= 1`. The postcondition of the procedure is `card(Content') = 0`. A valid loop invariant must ensure that executing the loop body in a state satisfying the invariant 1) does not violate the precondition of `removeFirst`, and 2) leads to a state that satisfies the loop invariant. A valid loop invariant must also ensure that upon termination of the loop, the postcondition of `clear` holds. One possible loop invariant that satisfies these criteria is $I_p : e' \Leftrightarrow \mathsf{card}(\texttt{Content'}) = 0$. Since e' is always false at the top of the loop body, I_p expresses the condition that the set is non-empty, thereby guaranteeing that the loop body can execute correctly; and since e' is always true when the loop exits, I_p implies that the set is empty at the end of the procedure, satisfying the procedure postcondition.

Our analysis plugin analyzes the `clear()` procedure by starting with the procedure precondition (in this case, `true`) and successively computing an approximation of the strongest postcondition over the statements in the procedure. Eventually, the analysis reaches `head`, the `while()` statement containing the loop, with the intermediate analysis result f. By construction, f holds for all reachable states at program counter `head` that the analysis has explored up to this point. In our example, f is the formula:

$$f = (\exists e_3.\ \neg e_3) \wedge q' = \emptyset \ \wedge\ (e' \Leftrightarrow \neg \mathsf{card}(\texttt{Content'}) \geq 1) \ \wedge\ \texttt{Content} = \texttt{Content'}$$

The formula f states that: 1) at some intermediate stage, the variable e was false (in this case, e was initially `false`); 2) the variable q points to null; 3) `e'` is true iff the `Content` set is nonempty; and 4) the `Content` set is unchanged from its value on entry to the procedure.

Our inference algorithm next strengthens f by conjoining the loop condition, producing a formula f_0 which holds at the start of the loop at the label `body` after zero loop iterations. For our example, f_0 is $f \wedge \neg e'$:

$$f_0 = (\exists e_3.\ \neg e_3) \wedge q' = \emptyset \wedge (e' \Leftrightarrow \neg \mathsf{card}(\texttt{Content'}) \geq 1) \wedge \texttt{Content} = \texttt{Content'} \wedge \neg e'$$

Since any loop invariant I must hold for all such states, it must be the case that $f_0 \Rightarrow I$. However, f_0 is unlikely to be the desired loop invariant, since it does not take into account the effect of the loop body. We therefore compute the strongest postcondition over the loop body, starting with f_0 at the top of the loop body, to obtain f_0'. The formula f_0' holds for the set of states that are reachable at the loop entry after executing exactly one loop iteration. Any acceptable loop invariant I must satisfy the constraints $f_0 \Rightarrow I$ and $f_0' \Rightarrow I$. For our example:

$$\begin{aligned} f_0' = \; & (\exists e_3.\ \neg e_3) \wedge (e' \Leftrightarrow \neg \mathsf{card}(\texttt{Content'}) \geq 1) \\ & \wedge\ (\exists e_5.\ \neg e_5 \wedge (e_5 \Leftrightarrow \mathsf{card}(\texttt{Content}) = 1)) \\ & \wedge\ \texttt{Content'} = \texttt{Content} \setminus q' \wedge \mathsf{card}(q') = 1 \wedge q' \in \texttt{Content} \wedge e' \end{aligned}$$

The formula f_0' states that the set `Content'` is equal to the set `Content` minus q', which points to an object in the heap (since $\mathsf{card}(q') = 1$). The formula f_0' also states that at some previous program state, the variable e was true iff the set `Content` had cardinality 1. (Note that e_5 was formerly e' at the top of the loop; the composition operation renames e' to the existentially quantified e_5.) Finally, f_0' states that at some previous program state, the variable e was false, and that at the present state, e is true iff the `Content'` set is empty; note that these final two conjuncts are common to f_0 and f_0'.

Building Potential Invariants. The formula f_0 summarizes the program state after zero iterations of the loop body; f_0' summarizes the state after one iteration. Our goal is to produce a logical formula which holds after an arbitrary number of loop iterations; we can start by producing a formula which holds after either zero or one loop iterations. We take conjuncts from f_0 which are implied by f_0', as well as conjuncts from f_0' which are implied by f_0. Any such conjuncts will then hold after both zero and one iterations of the loop body. We conjoin these conjuncts to produce the formula f_1:

$$\begin{aligned} f_1 = (\exists e_3.\ \neg e_3) \wedge (e' \Leftrightarrow \neg \mathsf{card}(\texttt{Content'}) \geq 1) \\ \wedge\ \texttt{Content'} = \texttt{Content} \setminus q' \wedge q' \in \texttt{Content} \end{aligned}$$

In formula f_1, we dropped the intermediate state e_5 and the constraint $\mathsf{card}(q') = 1$. The intermediate state e_5 was dropped because it does not exist after zero iterations of the loop. The cardinality constraint was dropped because q' is the empty set in f_0 and known to be nonempty in f_1. Dropping the cardinality constraint allows q' to contain an arbitrary number of heap objects; it is no longer required to point to a single location in the heap.

Our technique then checks whether f_1 is a loop invariant, using the technique described above for verifying explicit loop invariants. In our example, f_1 is not a loop invariant: it contains the conjunct $\texttt{Content'} = \texttt{Content} \setminus q'$, where q' is a free variable; that is, in all iterations of the loop, `Content'` is equal to `Content` minus q', for all values of q' (which is also constrained to be a subset of `Content`). While this conjunct holds for the zeroth and first iterations of the loop, it does not hold for all iterations of the loop. Therefore, we iterate again, computing f_1', the strongest postcondition of f_1 over the loop body. We combine conjuncts from f_1 which are implied by f_1' with conjuncts from f_1' which are implied by f_1, yielding the next estimate f_2.

The formula f_2 summarizes the program state after zero, one and two iterations. It contains the clause `Content'` $=$ `Content` $\setminus q_8 \setminus q'$. Because q_8 is existentially-quantified (rather than free), and because q_8 does not carry any cardinality constraints, the set q_8 can be interpreted to represent the difference between the initial `Content` set and the intermediate `Content'` set after any number of loop iterations. The analysis tests f_2 and finds that it is a loop invariant.

$$\begin{aligned} f'_1 = \;& \exists e_9.\, (\neg e_9 \wedge \exists q_8.\, (q_8 \in \texttt{Content} \wedge q' \in \texttt{Content} \setminus q_8 \\ & \qquad\qquad \wedge\, \texttt{Content'} = \texttt{Content} \setminus q_8 \setminus q') \\ & \qquad\qquad \wedge\, (\neg e_9 \Leftrightarrow \mathsf{card}(\texttt{Content} \setminus q_8) = 1)) \\ & \wedge\, (\exists e_3.\, \neg e_3) \wedge \mathsf{card}(\text{q'}) = 1 \wedge (e' \Leftrightarrow \neg\mathsf{card}(\texttt{Content'}) \geq 1) \end{aligned}$$

$$\begin{aligned} f_2 = \;& \exists q_8.\, (q_8 \in \texttt{Content} \wedge q' \in \texttt{Content} \setminus q_8 \\ & \qquad\qquad \wedge\, \texttt{Content'} = \texttt{Content} \setminus q_8 \setminus q') \\ & \wedge\, (\exists e_3.\, \neg e_3) \wedge q' \in \texttt{Content} \wedge (e' \Leftrightarrow \neg\mathsf{card}(\texttt{Content'}) \geq 1) \end{aligned}$$

Existential Quantifiers. In our exposition so far, we have ignored the internal structure of the conjuncts in our formulas, and treated each top-level conjunct as an atomic unit. However, we found it necessary in practice to decompose top-level conjuncts, retaining only the parts of the conjunct which are true. In particular, our algorithm is able to infer stronger invariants by examining the internal structure of existentially quantified clauses, rather than dropping the entire clause. For instance, in the formula above, if c_j is of the form $\exists e.\ \bigwedge c^j_k$, then we drop sub-conjuncts c^j_k that are not implied by f'_i. Note, however, that even if some set of sub-conjuncts K such that $c^j_k \in K$ are individually implied by f'_i, it does not necessarily follow that $f'_i \Rightarrow \bigwedge K$: in the presence of existential quantifiers, two sub-conjuncts may conspire to contradict the antecedent. If we do construct such a K which fails to imply f'_i, then we drop those conjuncts of K that mention e and try again.

Comparing our inferred loop invariant f_2 with the invariant I_p, we can observe that f_2 has a number of extraneous clauses (e.g. $q' \in \texttt{Content} \wedge (\exists e_3.\ \neg e_3)$, and also the clause containing q_8) which are not required to verify the loop or the procedure in general. We have found no simple way to produce automatically produce smaller invariants. One possible heuristic is to eliminate those conjuncts from an inferred loop invariant which are not required for the analysis of the loop body to go through. In our experience, this strategy generates invariants that are sound, but too weak to prove the postconditions of some procedures, so we do not apply it.

Enforcing Termination. As presented above, our algorithm for generating and checking trial loop invariants is not guaranteed to terminate; we can construct contrived examples on which our algorithm does not terminate. In practice, we are able to infer all loop invariants in our example programs in at most three iterations.

A small change to the algorithm presented above ensures termination in all cases where it is possible to construct a loop invariant. We limit the number of iterations that the original algorithm may execute. Once the limit is reached, the algorithm subsequently drops any non-preserved conjuncts and does not introduce any new ones; that is,

$$f_{i+1} = \bigwedge_j \{c_j \mid f_i' \Rightarrow c_j\}.$$

This phase is guaranteed to terminate because it operates on a finite number of conjuncts; no new conjuncts are added. If no conjuncts are dropped in a given iteration, then the algorithm has found a loop invariant and terminates. Otherwise, the size of the formula strictly decreases at each step.

Our algorithm, as amended, is guaranteed to never loop with an infinite sequence of potential invariants that are too strong. On the other hand, we can construct an example where our algorithm produces an invariant that is not strong enough for verifying the loop body. If a loop invariant exists, the developer can provide a hint to the inference algorithm by inserting the pair of statements `assert C; assume C;` inside the loop body.

Experience with Loop Invariants. We applied our loop invariant inference algorithm to our suite of benchmarks, which includes an HTTP server, a minesweeper implementation, and various small programs (see Section 8). Our inference algorithm successfully inferred all 15 invariants in our benchmark programs. In a previous version of our system [26], we used a simpler technique for loop invariant inference. The narrow applicability of our previous technique required us to manually supply loop invariants for most loops in our example programs. Because the manually written loop invariants were available to us, we were able to compare the developer-supplied loop invariants with the automatically inferred loop invariants. In all cases, the developer-supplied invariants are simpler than the inferred loop invariants, and the developer-supplied invariants implied the inferred loop invariants. The main sources of complexity in the inferred loop invariant are 1) the preservation of (an approximation of the) strongest postcondition throughout the loop, including set equalities between primed and unprimed sets; and 2) the introduction of existential quantifiers, as discussed above.

Discussion. We were surprised to discover that our simple loop invariant inference technique was able to infer all of the invariants in our example programs. Three properties of the Hob system seem to contribute to the feasibility of inferring loop invariants. In general, it seems that loop invariants are much easier to infer when the specification language is based on sets (contrast this to the JML specification language, which allows full Java expressions as specifications). The set specification language contributes to rich but focussed specifications for invoked procedures, which the loop inference algorithm can productively use to build its loop invariant, as we can observe in our example: the emptiness constraint on the `Content` set is the crucial ingredient in constructing the right invariant. Furthermore, the fact that formulas in our flag analysis are composed of a set of conjuncts (in part due to the manipulations described in Section 5) allows the loop invariant inference algorithm to drop some of the conjuncts as needed. Our experience reinforces our belief that a set-based specification language can give a valuable, high-level description of program behaviour, making program understanding easier for both programmers and programs.

7 Other Plugins

In addition to the flag analysis, we implemented a shape analysis plugin and a theorem proving plugin. These two plugins enable the Hob system to analyze complex properties of encapsulated data structures. To see the importance of these two plugins, note that the flag analysis captures the sharing of objects at the granularity of data structures represented as sets. This greatly simplifies and improves the scalability of the flag plugin. The reason that the flag analysis can reason in terms of abstract sets is that the other analyses verify that complex data structures are correctly represented using sets.

The shape analysis plugin enables precise verification of tree-based data structures. It uses a previously implemented tool, the Pointer Assertion Logic Engine [31] (PALE). We have incorporated PALE into our framework with essentially no changes to the tool itself[7]. The Hob framework effectively enabled the PALE tool to be applied to programs to which it was previously not applicable due to both scalability reasons and the limitations of the PALE programming model.

To verify even more detailed and precise data structure consistency properties, we implemented a theorem proving plugin [38]. The theorem proving plugin generates verification conditions suitable for interactive verification using the Isabelle proof assistant [32]. We successfully used the theorem proving plugin to verify array-based data structures such as a priority queue implemented as a binary heap.

8 Experience

We have implemented the Hob system, populated it with several analyses (including the flag, shape analysis, and theorem prover plugins), and used the system to develop several benchmark programs and applications. Figure 12 presents a subset of the benchmarks we ran through our system; full descriptions of our benchmarks (as well as the full source code for our modular pluggable analysis system) are available at our project homepage at `http://hob.csail.mit.edu`. Minesweeper, water and httpd are complete applications; the others are either computational patterns (compiler, scheduler, ctas) or data structures (prodcons). Compiler models a constant-folding compiler pass, scheduler models an operating system scheduler, and ctas models

[7] We modified PALE to indicate success or failure with an exit code.

	Number of modules	Lines of spec	Lines of impl
prodcons		41	50
compiler		86	143
scheduler		37	22
ctas		53	53
board		126	222
controller		76	155
view		57	358
minesweeper	6	367	889
atom		31	64
ensemble		164	883
h2o		158	423
water	10	582	1979
sendfile		32	162
httpserver		20	79
httprequest		49	128
httpd	10	246	614

Fig. 12. Benchmark characteristics

the core of an air-traffic control system. The board, controller, and view modules are the core minesweeper modules; atom, ensemble, and h2o are the core water modules; and sendfile, httpserver and httprequest the core httpd modules. The **bold** entries indicate system totals for minesweeper and water; note that minesweeper includes several other modules, some of which are analyzed by the shape analysis and theorem proving plugins, not the flag plugin.

We next present the impact of the formula transformation optimizations, then discuss the properties that we were able to specify and verify in the minesweeper and water benchmarks.

8.1 Formula Transformations

We analyzed our benchmarks on a 2.80GHz Pentium 4, running Linux, with 3 gigabytes of RAM. Figure 13 summarizes the results of our formula transformation optimizations. A ✓in the "Optimizations" column indicates a run in which all optimizations are enabled; an $\times$ indicates a run in which they are disabled. The "Number of nodes" column reports the sizes (in terms of AST node counts) of the resulting boolean algebra formulas. Our results indicate that the formula transformations reduce the formula size by 3.5 to greater than 80 times (often with greater reductions for larger formulas); the Optimization Ratio column presents the reduction obtained in formula size. The "MONA time" column presents the time spent in the MONA decision procedure (up to 87 seconds after optimization); the "Flag time" column presents the time spent in the flag analysis, excluding the decision procedure (up to 46 seconds after optimization). Without optimization, MONA could not successfully check the formulas for the compiler, board, view, ensemble and h2o modules because of an out of memory error.

8.2 Minesweeper

We next illustrate how our approach enables the verification of properties that span multiple modules. Our minesweeper implementation has several modules: a game board module (which represents the game state), a controller module (which responds to user input), a view module (which produces the game's output), an exposed cell module (which stores the exposed cells in an array), and an unexposed cell module (which stores the unexposed cells in an instantiated linked list). There are 787 non-blank lines of implementation code in the 6 implementation modules and 328 non-blank lines in the specification and abstraction modules.

Minesweeper uses the standard model-view-controller (MVC) design pattern [16]. The `board` module (which stores an array of `Cell` objects) implements the model part of the MVC pattern. Each `Cell` object may be mined, exposed or marked. The `board` module represents this state information using the `isMined`, `isExposed` and `isMarked` fields of `Cell` objects. At an abstract level, the sets `MarkedCells`, `MinedCells`, `ExposedCells`, `UnexposedCells`,

	Optimizations	Number of nodes	Optimization ratio	MONA time	Flag time
prodcons	✓	465	9.64	0.21	0.09
	×	4487	—	0.182	0.08
compiler	✓	5749	> 78.56	0.42	0.30
	×	> 451628	—	N/A	> 56.10
scheduler	✓	296	3.95	0.09	0.06
	×	1169	—	0.126	0.06
ctas	✓	3152	3.58	0.31	0.14
	×	11292	—	17.52	0.51
board	✓	11778	> 80.31	1.41	0.81
	×	> 945887	—	N/A	> 138.25
controller	✓	4528	6.38	0.69	0.19
	×	28904	—	3.20	0.80
view	✓	19311	N/A	5.45	1.48
	×	N/A	—	N/A	> 438.50
atom	✓	28317	20.65	5.33	0.74
	×	584834	—	1017.09	27.76
ensemble	✓	668110	N/A	86.99	46.05
	×	N/A	—	N/A	> 4070.00
h2o	✓	79249	> 15.87	15.96	25.70
	×	> 1257883	N/A	N/A	> 3282.08
sendfile	✓	2672	44.64	0.87	0.13
	×	119287	—	265.01	5.26
httpserver	✓	1094	62.34	0.36	0.21
	×	68198	—	36.68	5.75
httprequest	✓	9521	6.41	0.67	0.34
	×	61041	—	12.589	3.79

Fig. 13. Formula sizes before and after transformation

and `U` (for Universe) represent sets of cells with various properties; the `U` set contains all cells known to the board. The board also uses a global boolean variable `gameOver`, which it sets to `true` when the game ends.

Our system verifies that our implementation has the following properties (among others):

- The sets of exposed and unexposed cells are disjoint; unless the game is over, the sets of mined and exposed cells are also disjoint.
- The set of unexposed cells maintained in the `board` module is identical to the set of unexposed cells maintained in the `UnexposedList` list.
- The set of exposed cells maintained in the `board` module is identical to the set of exposed cells maintained in the `ExposedSet` array.
- At the end of the game, all cells are revealed; *i.e.* the set of unexposed cells is empty.

Although our system focuses on using sets to model program state, not every module needs to define its own abstract sets. Indeed, certain modules may not define any abstract sets of their own, but instead coordinate the activity of other modules to accomplish tasks. The `view` and `controller` modules are examples of such modules. The `view` module has no state at all; it queries the board for the current game state and calls the system graphics libraries to display the state.

Because these modules coordinate the actions of other modules—and do not encapsulate any data structures of their own—the analysis of these modules must operate solely at the level of abstract sets. Our analysis is capable of ensuring the validity of these modules, since it can track abstract set membership, solve formulas in the boolean algebra of sets, and incorporate the effects of invoked procedures as it analyzes each module. Note that for these modules, our analysis need not reason about any correspondence between concrete data structure representations and abstract sets.

The set abstraction supports typestate-style reasoning at the level of individual objects (for example, all objects in the `ExposedCells` set can be viewed as having a conceptual typestate `Exposed`). Our system also supports the notion of global typestate. The `board` module, for example, has a global `gameOver` variable which indicates whether or not the game is over. The system uses this variable and the definitions of relevant sets to maintain the global invariant `gameOver | disjoint(MinedCells, ExposedCells)`.

This global invariant connects a global typestate property—is the game over?—with a object-based typestate state property evaluated on objects in the program—there are no mined cells that are also exposed. Our analysis plugins verify these global invariants by conjoining them to the preconditions and postconditions of methods. Note that global invariants must be true in the initial state of the program. If some initializer must execute to establish an invariant, then the invariant can be guarded by a global typestate variable.

Another invariant concerns the correspondence between the `ExposedCells`, `UnexposedCells`, `ExposedSet.Content`, and `UnexposedList.Content` sets:

```
(ExposedCells = ExposedSet.Content) & (UnexposedCells = UnexposedList.Content)
```

Our analysis verifies this property by conjoining it to the `ensures` and `requires` clauses of the appropriate procedures. The `board` module is responsible for maintaining this invariant. Yet the analysis of the board module does not, in isolation, have the ability to completely verify the invariant: it cannot reason about the concrete state of `ExposedSet.Content` or `UnexposedList.Content` (which are defined in other modules). However, the `ensures` clauses of its callees, in combination with its own reasoning that tracks membership in the `ExposedCells` set, enables our analysis to verify the invariant (assuming that `ExposedSet` and `UnexposedList` work correctly).

Our system found a number of errors during the development and maintenance of our minesweeper implementation. We next present one of these errors. At the end of the game, minesweeper exposes the entire game board; we use `removeFirst` to remove all elements from the unexposed list, one at a time. After we have exposed the entire board, we can guarantee that the list of unexposed cells is empty:

```
proc drawFieldEnd()
    requires ExposedList.setInit & Board.gameOver &
             (UnexposedList.Content <= Board.U)
    modifies UnexposedList.Content, Board.ExposedCells,
              Board.UnexposedCells, ExposedList.Content,
              UnexposedList.Content
    ensures card(UnexposedList.Content') = 0;
```

because the implementation of the `drawFieldEnd` procedure loops until `isEmpty` returns `true`, which also guarantees that the `UnexposedList.Content` set is empty. The natural way to write the iteration in this procedure would be:

```
while (UnexposedList.isEmpty()) {
    Cell c = UnexposedList.removeFirst();
    drawCellEnd(c);
}
```

and indeed, this was the initial implementation of that code. However, when we attempted to analyze this code, we got the following error message:

```
Analyzing proc drawFieldEnd...
Error found analyzing procedure drawFieldEnd:
    requires clause in a call to procedure View.drawCellEnd.
```

Upon further examination, we found that we were breaking the invariant `Board.ExposedCells` = `UnexposedList.Content`. The correct way to preserve the invariant is by calling `Board.setExposed`, which simultaneously sets the `isExposed` flag and removes the cell from the `UnexposedList`:

```
Cell c = UnexposedList.getFirst();
Board.setExposed(c, true);
drawCellEnd(c);
```

8.3 Water

Water is a port of the Perfect Club benchmark MDG [2]. It uses a predictor/corrector method to evaluate forces and potentials in a system of water molecules in the liquid state. The central loop of the computation performs a time step simulation. Each step predicts the state of the simulation, uses the predicted state to compute the forces acting on each molecule, uses the computed forces to correct the prediction and obtain a new simulation state, then uses the new simulation state to compute the potential and kinetic energy of the system.

Water consists of several modules, including the `simparm`, `atom`, `H2O`, `ensemble`, and `main` modules. These modules contain 2000 lines of implementation and 500 lines of specification. Each module defines sets and boolean variables; we use these sets and variables to express safety properties about the computation.

The `simparm` module, for instance, is responsible for recording simulation parameters, which are stored in a text file and loaded at the start of the computation. This module defines two boolean variables, `Init` and `ParmsLoaded`. If `Init` is true, then the module has been initialized, *i.e.* the appropriate arrays have been allocated on the heap. If `ParmsLoaded` is true, then the simulation parameters have been loaded from disk and written into these arrays. Our analysis verifies that the program does not load simulation parameters until the arrays have been allocated and does not read simulation parameters until they have been loaded from the disk and written into the arrays.

The fundamental unit of the simulation is the atom, which is encapsulated within the `atom` module. Atoms cycle between the *predicted* and *corrected* states, with the `predic` and `correc` procedures performing the computations necessary to effect these state changes. A correct computation will only predict a corrected atom or correct a predicted atom. To enforce this property, we define two sets `Predic` and `Correc` and populate them with the predicted and corrected atoms, respectively. The `correc` procedure operates on a single atom; its precondition requires this atom to be a member of the `Predic` set. Its postcondition ensures that, after successful completion, the atom is no longer in the `Predic` set, but is instead in the `Correc` set. The `predic` procedure has a corresponding symmetric specification.

Atoms belong to molecules, which are handled by the `H2O` module. A molecule tracks the position and velocity of its three atoms. Like atoms, each module can be in a variety of conceptual states. These states indicate not only whether the program has predicted or corrected the position of the molecule's atoms but also whether the program has applied the intra-molecule force corrections, whether it has scaled the forces acting on the molecule, etc. We verify the invariant that when the molecule is in the predicted or corrected state, the atoms in the molecule are also in the same state. The interface of the `H2O` module ensures that the program performs the operations on each molecule in the correct order — for example, the `bndry` procedure may operate only on molecules in the `Kineti` set (which have had their kinetic energy calculated by the `kineti` procedure).

The `ensemble` module manages the collection of molecule objects. This module stages the entire simulation by iterating over all molecules and computing their positions and velocities over time. The ensemble module uses boolean predicates to track the state of the computation. When the boolean predicate `INTERF` is true, for example, then the program has completed the interforce computation for all molecules in the simulation. Our analysis verifies that the boolean predicates, representing program state, satisfy the following ordering relationship:

$$\texttt{Init} \leadsto \texttt{INITIA} \leadsto \texttt{PREDIC} \leadsto \texttt{INTRAF} \leadsto \texttt{VIR} \leadsto \texttt{INTERF} \leadsto \cdots$$

Our specification relies on an implication from boolean predicates to properties ranging over the collection of molecule objects, which can be ensured by a separate array analysis plugin [24].

These properties help ensure that the computation's phases execute in the correct order; they are especially valuable in the maintenance phase of a program's life, when the original designer, if available, may have long since forgotten the program's phase ordering constraints. Our analysis' set cardinality constraints also prevent empty sets (and null pointers) from being passed to procedures that expect non-empty sets or non-null pointers.

9 Related Work

In this section we discuss related work in the general area of program checking tools and other typestate systems in particular. We start by comparing the Hob framework to the approach taken by the ESC/Java and Boogie program checking tools; next, we discuss general properties of typestate systems and compare Hob's sets to typestate systems.

Program checking tools. ESC/Java [14] is a program checking tool whose purpose is to identify common errors in programs using program specifications in a subset of the Java Modelling Language (JML) [3]. ESC/Java sacrifices soundness in that it does not model all details of the program heap, but can detect some common programming errors. The Spec$^{\#}$ programming system [1] adds similar features to C$^{\#}$, including the ability to specify method contracts, frame conditions and class contracts. These contracts may be verified at runtime or by the Boogie static verifier, which uses a theorem prover to discharge its verification conditions.

We discuss two key differences between our approach and the proposed Boogie approach. First, Boogie envisions the use of a single general-purpose theorem prover to discharge the generated verification conditions. Hob, on the other hand, is designed to support a diverse range of potentially narrow, specialized analyses (this range includes shape analyses, typestate analyses [26] and even interactive theorem provers [38] as well as less detailed analyses). This goal is reflected in Hob's format construct and in its abstract set specification language, both of which are designed to support a strong separation between different analyses (such a separation is necessary, of course, if multiple analyses are to cooperate to successfully analyze a single program). This approach minimizes the amount of expertise required to work within the Hob system and maximizes the ability of developers with specialized skills to contribute. We believe that enabling as many developers to contribute as possible will lead to a richer, more powerful analysis system.

Second, Boogie is designed to verify object invariants, with an object ownership mechanism supporting the hierarchical specification and verification of invariants that involve hierarchies of linked objects. This mechanism eliminates a form of specification aggregation for computations that traverse a hierarchy of owned objects—if the procedure call hierarchy matches the ownership hierarchy, each procedure need only state consistency requirements for the object that it directly accesses, not all of the child objects that that object owns. This hierarchical specification approach is reminiscent of hierarchical access specifications in Jade [33] and hierarchical locking mechanisms in databases [36].

Hob, on the other hand, is designed to support computations organized around a flat set of data structures. The constructs that eliminate specification aggregation cut across the procedure call hierarchy rather than working within it. This adoption of cross-cutting organizational approaches reflects the maturation of computer science as a discipline—over time, the overwhelming dominance of hierarchical approaches will fade as the effectiveness of using other approaches in addition to hierarchies becomes obvious.

Typestate systems. Typestate systems track the conceptual states that each object goes through during its lifetime in the computation [7, 9–12, 37]. They generalize standard type systems in that the typestate of an object may change during the computation. Aliasing (or more generally, any kind of sharing) is the key problem for typestate systems—if the program uses one reference to change the typestate of an object, the typestate system must ensure that either the declared typestate of the other references is updated to reflect the new typestate or that the new typestate is compatible with the old declared typestate at the other references.

Most typestate systems avoid this problem altogether by eliminating the possibility of aliasing [37]. Generalizations support monotonic typestate changes (which ensure that the new typestate remains compatible with all existing aliases) [12] and enable the program to temporarily prevent the program from using a set of potential aliases, change the typestate of an object with aliases only in that set, then restore the typestate and reenable the use of the aliases [10]. It is also possible to support object-oriented constructs such as inheritance [8]. Finally, in the role system, the declared typestate of each object characterizes all of the references to the object, which enables the typestate system to check that the new typestate is compatible with all remaining aliases after a nonmonotonic typestate change [21].

In our approach, the typestate of each object is determined by its membership in abstract sets as determined by the values of its encapsulated fields and its participation in encapsulated data structures. Our system supports generalizations of the standard typestate approach such as orthogonal typestate composition and hierarchical typestate classification. The connection with data structure participation enables the verification of both local and global data structure consistency properties.

10 Conclusion

Typestate systems have traditionally been designed to enforce safety conditions that involve objects whose state may change during the course of the computation. In particular, the standard goal of typestate systems is to ensure that operations are invoked only on objects that are in appropriate states. Most existing typestate systems support a flat set of object states and limit typestate changes in the presence of sharing caused by aliasing. We have presented a reformulation of typestate systems in which the typestate of each object is determined by its membership in abstract typestate sets. This reformulation supports important generalizations of the typestate concept such as typestates that capture membership in data structures, composite typestates in which objects are members of multiple typestate sets, hierarchical typestates, and cardinality constraints on the number of objects that are in a given typestate. In the context of our Hob modular pluggable analysis framework, our system also enables the specification and effective verification of detailed local and global data structure consistency properties, including arbitrary internal consistency properties of linked and array-based data structures. Our system therefore effectively supports tasks such as understanding the global sharing patterns in large programs, verifying the absence of undesirable interactions, and ensuring the preservation of critical properties necessary for the correct operation of the program.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS: Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[2] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, Nov. 1992.

[3] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. Technical Report NII-R0309, Computing Science Institute, Univ. of Nijmegen, March 2003.

[4] L. Clarke and D. Richardson. Symbolic evaluation methods for program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 9. Prentice-Hall, Inc., 1981.

[5] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th POPL*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[7] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.

[8] R. DeLine and M. Fähndrich. Typestates for objects. In *Proc. 18th ECOOP*, June 2004.

[9] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proc. 15th ECOOP*, LNCS 2072, pages 130–149. Springer, 2001.

[10] M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. ACM PLDI*, 2002.

[11] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 302–312. ACM Press, 2003.

[12] M. Fähndrich and K. R. M. Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.

[13] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*. Springer, 2003.

[14] C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2002.

[15] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proc. 28th ACM POPL*, 2001.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlisside. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., 1994.

[17] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *11th SAS*, 2004.

[18] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual.* BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[19] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata.* LNCS, 2000.

[20] D. Kozen. Complexity of boolean algebras. *Theoretical Computer Science*, 10:221–247, 1980.

[21] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Annual ACM Symp. on Principles of Programming Languages (POPL)*, 2002.

[22] V. Kuncak, H. H. Nguyen, and M. Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *20th International Conference on Automated Deduction, CADE-20*, Tallinn, Estonia, July 2005.

[23] V. Kuncak and M. Rinard. The first-order theory of sets with cardinality constraints is decidable. Technical Report 958, MIT CSAIL, July 2004.

[24] P. Lam, V. Kuncak, and M. Rinard. On our experience with modular pluggable analyses. Technical Report 965, MIT CSAIL, September 2004.

[25] P. Lam, V. Kuncak, and M. Rinard. Cross-cutting techniques in program specification and analysis. In *4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, 2005.

[26] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[27] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *14th International Conference on Compiler Construction (tool demo)*, April 2005.

[28] P. Lam, V. Kuncak, K. Zee, and M. Rinard. The Hob project web page. http://hob.csail.mit.edu, 2004.

[29] K. R. M. Leino. Efficient weakest preconditions. KRML114a, 2003.

[30] L. Loewenheim. Über Mögligkeiten im Relativkalkül. *Math. Annalen*, 76:228–251, 1915.

[31] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.

[32] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[33] M. C. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford University, 1994.

[34] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[35] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis problems. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981.

[36] A. Silberschatz and Z. Kedem. Consistency in hierarchical database systems. *Journal of the ACM*, 27(1):72–80, January 1980.

[37] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, January 1986.

[38] K. Zee, P. Lam, V. Kuncak, and M. Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, November 2004.

A Transfer Functions

This section presents the transfer functions for the flag analysis.

Assignment statements. We first define a generic frame condition generator, used in our transfer functions,

$$\mathsf{frame}_x = \bigwedge_{S \neq x,\ \text{S not derived}} S' = S \wedge \bigwedge_{p \neq x} (p' \Leftrightarrow p),$$

where S ranges over sets and p over boolean predicates. Note that derived sets are not preserved by frame conditions; instead, the analysis preserves the anonymous sets contained in the derived set definitions and conjoins these definitions to formulas before applying the decision procedure.

Our flag analysis also tracks values of boolean variables:

$$\begin{aligned}
\mathcal{F}(\mathtt{b} = \mathsf{true}) &= \mathtt{b}' \wedge \mathsf{frame}_b \\
\mathcal{F}(\mathtt{b} = \mathsf{false}) &= (\neg \mathtt{b}') \wedge \mathsf{frame}_b \\
\mathcal{F}(\mathtt{b} = \mathtt{y}) &= (\mathtt{b}' \Leftrightarrow \mathtt{y}) \wedge \mathsf{frame}_b \\
\mathcal{F}(\mathtt{b} = \langle \text{if cond} \rangle) &= (\mathtt{b}' \Leftrightarrow f^{+}(\langle \text{if cond} \rangle)) \wedge \mathsf{frame}_b \\
\mathcal{F}(\mathtt{b} = !e) &= \mathcal{F}(\mathtt{b} = e) \circ ((\mathtt{b}' \Leftrightarrow \neg \mathtt{b}) \wedge \mathsf{frame}_b)
\end{aligned}$$

where $f^{+}(e)$ is the result of evaluating e, defined below in our analysis of conditionals.

The analysis also track local variable object references:

$$\mathcal{F}(\mathtt{x} = \mathtt{y}) = (\mathtt{x}' = \mathtt{y}) \wedge \mathsf{frame}_x \qquad \mathcal{F}(\mathtt{x} = \mathsf{null}) = (\mathtt{x}' = \emptyset) \wedge \mathsf{frame}_x$$
$$\mathcal{F}(\mathtt{x} = \mathsf{new}\ t) = \neg(\mathtt{x}' = \emptyset) \wedge \textstyle\bigwedge_S (\mathtt{x}' \cap S = \emptyset) \wedge \mathsf{frame}_x$$

We next present the transfer function for changing set membership. If $R = \{x : T \mid x.\mathtt{f} = c\}$ is a set definition in the abstraction section, we have:

$$\mathcal{F}(\mathtt{x.f} = \mathtt{c}) = R' = R \cup \mathtt{x} \wedge \bigwedge_{S \in \mathsf{alts}(R)} S' = S \setminus \mathtt{x} \wedge \ \mathsf{frame}_{\{R\} \cup \, \mathsf{alts}(R)}$$

where $\mathsf{alts}(\mathtt{R}) = \{S \mid$ abstraction module contains $S = \{x : T \mid x.\mathtt{f} = c_1\}, c_1 \neq c.\}$

The rules for reads and writes of boolean fields are similar but, because our analysis tracks the flow of boolean values, more detailed:

$$\begin{array}{l} \mathcal{F}(\mathtt{x.f} = \mathtt{b}) = \left(\begin{array}{l} b \wedge {B^{+}}' = B^{+} \cup \mathtt{x} \\ \quad \wedge \bigwedge_{S \in \mathsf{alts}(\mathtt{B}^{+})} S' = S \setminus \mathtt{x} \end{array} \right) \wedge \left(\begin{array}{l} \neg b \wedge {B^{-}}' = B^{-} \cup \mathtt{x} \\ \quad \wedge \bigwedge_{S \in \mathsf{alts}(\mathtt{B}^{-})} S' = S \setminus \mathtt{x} \end{array} \right) \\ \qquad\qquad \wedge \mathsf{frame}_{\{B\} \cup \mathsf{alts}(B)} \\ \mathcal{F}(\mathtt{b} = \mathtt{y.f}) = (b' \Leftrightarrow y \in B^{+}) \wedge \mathsf{frame}_{b}. \end{array}$$

where $B^{+} = \{x : T \mid x.\mathtt{f} = \mathsf{true}\}$ and $B^{-} = \{x : T \mid x.\mathtt{f} = \mathsf{false}\}$.

Finally, we have some default rules to conservatively account for expressions not otherwise handled,

$$\mathcal{F}(x.f = *) = \mathsf{frame}_x \qquad \mathcal{F}(x = *) = \mathsf{frame}_x.$$

Procedure calls. For a procedure call `x=proc(y)`, our transfer function checks that the callee's requires condition holds, then incorporates `proc`'s ensures condition as follows:

$$\mathcal{F}(\mathtt{x} = \mathtt{proc}(\mathtt{y})) = \mathsf{ensures}_1(\mathtt{proc}) \wedge \bigwedge_{S} S' = S$$

where both $\mathsf{ensures}_1$ and $\mathsf{requires}_1$ substitute caller actuals for formals of `proc` (including the return value), and where S ranges over all local variables.

Conditionals. The analysis produces a different formula for each branch of an `if` statement `if (e)`. We define functions $f^{+}(e), f^{-}(e)$ to summarize the additional information available on each branch of the conditional; the transfer functions for the true and false branches of the conditional are thus, respectively,

$$[\![\mathtt{if\ (e)}]\!]^{+}(B) = f^{+}(e) \wedge B \qquad [\![\mathtt{if\ (e)}]\!]^{-}(B) = f^{-}(e) \wedge B.$$

For constants and logical operations, we define the obvious f^{+}, f^{-}:

$$\begin{array}{rclrcl} f^{+}(\mathsf{true}) & = & \mathsf{true} & f^{-}(\mathsf{true}) & = & \mathsf{false} \\ f^{+}(\mathsf{false}) & = & \mathsf{false} & f^{-}(\mathsf{false}) & = & \mathsf{true} \\ f^{+}(!e) & = & f^{-}(e) & f^{-}(!e) & = & f^{+}(e) \\ f^{+}(x\,\mathtt{!=}e) & = & f^{-}(x\mathtt{==}e) & f^{-}(x\,\mathtt{!=}e) & = & f^{+}(x\mathtt{==}e) \\ f^{+}(e_1\ \mathtt{\&\&}\ e_2) & = & f^{+}(e_1) \wedge f^{+}(e_2) & f^{-}(e_1\ \mathtt{\&\&}\ e_2) & = & f^{-}(e_1) \vee f^{-}(e_2) \end{array}$$

We define f^{+}, f^{-} for boolean fields as follows:

$$\begin{array}{rclrcl} f^{+}(x.f) & = & x \subseteq B & f^{-}(x.f) & = & x \not\subseteq B \\ f^{+}(x.f\mathtt{==}\mathsf{false}) & = & x \not\subseteq B & f^{-}(x.f\mathtt{==}\mathsf{false}) & = & x \subseteq B \end{array}$$

where $B = \{x : T \mid x.\mathtt{f} = \mathsf{true}\}$; analogously, let $R = \{x : T \mid x.\mathtt{f} = \mathsf{c}\}$. Then,

$$f^{+}(x.f\texttt{==}\mathsf{c}) = x \subseteq R \qquad f^{-}(x.f\texttt{==}\mathsf{c}) = x \not\subseteq R.$$

We also predicate the analysis on whether a reference is `null` or not:

$$f^{+}(x\texttt{==}\mathsf{null}) = x = \emptyset \qquad f^{-}(x\texttt{==}\mathsf{null}) = x \neq \emptyset.$$

Finally, we have a catch-all condition,

$$f^{+}(*) = \mathsf{true} \quad f^{-}(*) = \mathsf{true}$$

which conservatively captures the effect of unknown conditions.

Assertions and Assume Statements. We analyze statement s of the form `assert A` by showing that the formula for the program point s implies `A`. Assertions allow developers to check that a given set-based property holds at an intermediate point of a procedure. Using `assume` statements, we allow the developer to specify properties that are known to be true, but which have not been shown to hold by this analysis. Our analysis prints out a warning message when it processes `assume` statements, and conjoins the assumption to the current dataflow fact. Assume statements have proven to be valuable in understanding analysis outcomes during the debugging of procedure specifications and implementations. Assume statements may also be used to communicate properties of the implementation that go beyond the abstract representation used by the analysis.

Return Statements. Our analysis processes the statement `return x` as an assignment `rv = x`, where `rv` is the name given to the return value in the procedure declaration. For all return statements (whether or not a value is returned), our analysis checks that the current formula implies the procedure's postcondition and stops propagating that formula through the procedure.