Verifying Complex Properties using Symbolic Shape Analysis

Thomas Wies

Departement of Computer Science
University of Freiburg, Germany

Viktor Kuncak Karen Zee Martin Rinard

MIT, CSAIL
Cambridge, USA

Andreas Podelski

Departement of Computer Science
University of Freiburg, Germany

Abstract

One of the main challenges in the verification of software systems is the analysis of statically unbounded data structures with dynamic memory allocation, such as linked data structures and arrays. We describe Bohne, a new analysis for verifying data structures. Bohne verifies data structure operations and shows that 1) the operations preserve data structure invariants and 2) the operations satisfy their specifications expressed in terms of changes to the set of objects stored in the data structure. During the analysis, Bohne infers loop invariants in the form of disjunctions of universally quantified Boolean combinations of formulas, represented as sets of binary decision diagrams. To synthesize loop invariants of this form, Bohne uses a combination of decision procedures for Monadic Second-Order Logic over trees, SMT-LIB decision procedures (currently CVC Lite), first-order provers such as SPASS and E, and the automated reasoner within the Isabelle interactive theorem prover. This architecture shows that synthesized loop invariants can serve as a useful communication mechanism between different decision procedures. In addition, Bohne uses field constraint analysis, a combination mechanism that enables the use of uninterpreted function symbols within formulas of Monadic Second-Order Logic over trees. Using Bohne, we have verified operations on data structures such as linked lists with iterators and back pointers, trees with and without parent pointers, two-level skip lists, array data structures, and sorted lists. We have deployed Bohne in the Hob and Jahob data structure analysis systems, enabling us to combine Bohne with analyses of data structure clients and apply it in the context of larger programs. This paper describes the Bohne algorithm, the techniques that Bohne uses to reduce the amount of annotations and the running time of the analysis.

1 Introduction

Complex data structure invariants are one of the main challenges in verifying software systems. Unbounded data structures such as linked data structures and dynamically allocated arrays make the state space of software artifacts infinite and require new reasoning techniques (such as reasoning about reachability) that have traditionally not been part of theorem provers specialized for program verification. The ability of linked structures to change their shape makes them a powerful programming construct, but at the same time makes them difficult to analyze, because the appropriate analysis representation is dependent on the invariants that the program maintains. It is therefore not surprising that the most successful verification approaches for analysis of data structures use parameterized abstract domains; these analyses include parametric shape analysis [47] as well as predicate abstraction [3, 23] and its generalizations [14, 31].

This paper presents *Bohne*, an algorithm for inferring loop invariants of programs that manipulate heap-allocated data structures. Like predicate abstraction, Bohne is parameterized by the properties to be verified. What makes the Bohne algorithm unique is the use of a precise abstraction domain that can express detailed properties of different regions of a program's infinite memory, and a range of techniques for exploring this analysis domain

using decision procedures. The algorithm was initially developed as a symbolic shape analysis [53,43] for linked data structures and uses the key idea of many shape analyses, made explicit in the TVLA analyzer [47, 36]: the partitioning of objects according to certain unary predicates. One of the observations of our paper is that the synthesis of heap partitions is not only useful for analyzing shape properties (which involve transitive closure), but also for combining such shape properties with sorting properties of data structures and properties expressible using linear arithmetic and first-order logic.

1.1 Related Work

We next put the Bohne algorithm in the context of two abstract interpretation [11] approaches that are closest to symbolic shape analysis: predicate abstraction and parametric shape analysis. We then discuss the work on decision procedures because Bohne uses a validity checker for an expressive logic to perform the analysis.

Predicate abstraction. Bohne builds on predicate abstraction but introduces important new techniques that make it applicable to the domain of shape analysis. There are two main sources of complexity of loop invariants in shape analysis. The first source of complexity is the fact that the invariants contain reachability predicates. To address this problem, Bohne uses a decision procedure for monadic second-order logic over trees [50,25], and combines it with uninterpreted function symbols in a way that preserves completeness in important cases [54]. The second source of complexity is that the invariants contain universal quantifiers in an essential way. Among the main approaches for dealing with quantified invariants in predicate abstraction is the use of Skolem constants [14], indexed predicates [31] and the use of abstraction predicates that contain quantifiers. The key difficulty in using Skolem constants for shape analysis is that the properties of individual objects depend on the "context", given by the properties of surrounding objects, which means that it is not enough to use a fixed Skolem constant throughout the analysis; it is instead necessary to instantiate universal quantifiers from previous loop iterations, in some cases multiple times. Compared to indexed predicates [31] the domain used by Bohne is more general because it contains disjunctions of universally quantified statements. The presence of disjunctions is not only more expressive in principle, but allows Bohne to keep formulas under the universal quantifiers more specific. This enables the use of less precise, but more efficient algorithms for computing changes to properties of objects without losing too much precision in the overall analysis. Finally, the advantage of using abstraction tailored to shape analysis compared to using quantified global predicates is that the parameters to shape-analysis-oriented abstraction are properties of objects in a state, as opposed to global properties of a state, and the number of global predicates needed to emulate state predicates is exponential in the number of properties [39,53].

The advantages of combining predicate abstraction with shape analysis are clearly demonstrated in lazy shape analysis [6]. Lazy shape analysis performs independent runs of a shape analysis algorithm, whose results are then used to improve the precision of predicate abstraction. In contrast, our symbolic shape analysis generalizes predicate abstraction technique to the point where it itself becomes effective as a shape analysis. We note that Bohne has also been extended to perform the automated discovery of predicates; the discussion of this extension is beyond the scope of this paper.

Shape analysis. Shape analyses are precise analyses for linked data structures. They were

originally used for compiler optimizations [24, 19, 18] and lacked the precision needed to establish invariants that Bohne is analyzing. Precise data structure analyses for verification include [29, 16, 26, 35, 41, 20, 47] and have recently also been applied to verify set implementations [45]. Unlike Bohne, most shape analyses that synthesize loop invariants are based on precomputed transfer functions and a fixed (though parameterized) set of properties to be tracked; recent approaches enable automation of such computation using decision procedures [57, 56, 58, 43, 54] or finite differencing [46]. Our approach differs from [32] in using complete reasoning about reachability in both lists and trees, and using a different architecture of the reasoning procedure. Our reasoning procedure uses a coarse-grain combination of reachability reasoning with decision procedures and theorem provers for numerical and first-order properties, as opposed to using a Nelson-Oppen style theorem prover. This allowed us to easily combine several tools that were developed completely independently [25, 4, 42]. Shape analysis approaches have also been used to verify sortedness properties [38] relying on manually abstracting a sortedness relation.

Decision procedures. Our symbolic shape analysis algorithm relies on decision procedures for expressive logics to perform synthesis of loop invariants. The system then verifies that the synthesized invariants are sufficient to prove the absence of errors and to prove the postcondition. During the invariant synthesis, the analysis primarily uses the MONA decision procedure [25] with field constraint analysis [54] to reason about expressive invariants involving reachability in tree-like linked structures. Our analysis also uses CVC Lite [4] via the SMT-LIB interface to reason about array data structures, local properties in non-tree data structures, and linear arithmetic. These two decision procedures are most relevant for the present paper. In our system (Figure 1) we also use interactive theorem provers Isabelle [42] and Coq [5] to debug the proof obligations and translations into decision procedures, as well as to automatically discharge some proof obligations using simplication and proof search built into these provers. We have also had success using first-order theorem provers Vampire [51], E [48] (via the TPTP interface [49]) as well as SPASS [52]. We used first-order theorem provers in Jahob to verify implementations of data structures [8], avoiding the use of reachability using specification variables similarly to the approaches taken in [29, 40] and automated to some extent in [37]. Finally, to reason about the sizes of data structures, we used a new decision procedure [30, 28] with a reduction to Presburger arithmetic.

Several recent decision procedures address specifically linked lists [1, 12, 39, 7, 44], where the emphasis is on the predictability (decision procedures for well-defined classes of properties of linked lists), the efficiency (membership in NP), the ability to interoperate with other reasoning procedures, and modularity. Although the Bohne approach is not limited to lists, it can take advantage of decision procedures for lists by applying such specialized procedures when they are applicable and using more general reasoning otherwise. In our current experience, the MONA decision procedure [25] proved to be effective for verifying reachability in both list and tree structures.

Bohne could also take advantage of logics for reasoning about reachability, such as the logic of reachable shapes [55]. Existing logics, such as guarded fixpoint logic [21] and description logics with reachability [10, 17] are attractive because of their expressive power, but so far no decision procedures for these logics have been implemented.

1.2 Contributions

We have previously described the general idea of symbolic shape analysis [43] as well as the field constraint analysis decision procedure for combining reachability reasoning with uninterpreted function symbols [54]. These previous techniques are our starting point. The main contributions of this paper are the following:

(i) We describe a method for synthesis of Boolean heap programs that improves the efficiency of fixpoint evaluation by precomputing abstract transition relations and can control the precision/efficiency trade-off by recomputing transition relations on-demand during fixpoint computation.

(ii) We introduce semantic caching of decision procedure queries across different fixpoint iterations and even different analyzed procedures. The caching yields substantial improvements for procedures that exhibit some similarity, which opens up the possibility of using our analysis in an interactive context.

(iii) We describe a static analysis that propagates precondition conjuncts and quickly finds many true facts, reducing the running time and the number of needed abstraction predicates for the subsequent symbolic shape analysis.

(iv) We present a domain-specific quantifier instantiation technique that significantly improves the running time of the analysis. Furthermore, it often eliminates the need for the underlying decision procedures to deal with quantifiers.

Together, these new techniques allowed us to verify a range of data structures without specifying loop invariants and without specifying a large number of abstraction predicates. Our examples include implementations of lists (with iterators and with back pointers), trees with parent pointers, two-level skip lists, sorted lists, as well as combinations of these data structures. What makes these results particularly interesting is a higher level of automation than in previous approaches: Bohne synthesizes loop invariants that involve reachability expressions and numerical quantities, yet it does not have precomputed transfer functions for a particular set of abstraction predicates. Bohne instead uses decision procedures to reason about arbitrary predicates definable in a given logic. Moreover, in our system the developer is not required to manually specify the changes of membership of elements in sets because such changes are computed by the system. Our system uses such synthesized invariants to communicate the information between different decision procedures.

Bohne as a component of Jahob. Bohne is part of the data structure verification frameworks Jahob [27, 28] and Hob [34, 33]. The goal of these systems is to verify data structure consistency properties in the context of non-trivial programs. To achieve this goal, these tools combine multiple static analyses, theorem proving, and decision procedures. In this paper we present our experience in deploying Bohne in the Jahob framework. The input language for Jahob is a subset of Java extended with annotations written as special comments. Therefore, Jahob programs can be compiled and executed using existing Java compilers and virtual machines.

Figure 1 illustrates the integration of Bohne into the Jahob framework. Bohne uses Jahob's facilities for symbolic execution of program statements and the validity checker to compute the abstraction of the source program. The output of Bohne is the source program annotated with the inferred loop invariants. The annotated program serves as an input to a verification condition generator. The generated verification conditions are verified using an

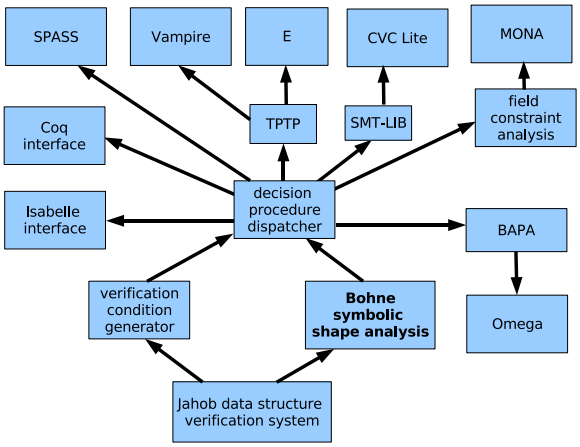

Fig. 1. Jahob Data Structure Analysis System Architecture

approach [28] that combines special purpose decision procedures, general purpose theorem provers, and reasoning techniques such as field constraint analysis [54].

2 Motivating Example

We illustrate our technique on the procedure `SortedList.insert` shown in Figure 2. This procedure inserts a `Node` object into a global sorted list. The annotation given by special comments `/*: ... */` consists of data structure invariants, pre- and postconditions, as well as hints for the analysis. Formulas are expressed in a subset of the language used in the Isabelle interactive theorem prover [42]. The specification uses an abstract set variable `content` which is defined as the set of non-null objects reachable from the global variable `first` by following the field `next`. The data structure invariants are specified by the annotation `invariant "..."`. For instance, the first invariant expresses the fact that the field `next` forms trees in the heap, i.e. that `next` is acyclic and injective; the third invariant expresses the fact that the elements stored in the list are sorted in increasing order according to field `data`. The precondition of the procedure, `requires "..."`, states that the object to be inserted is non-null and not yet contained in the list. The postcondition, `ensures "..."`, expresses that the argument is properly inserted into the list.

The loop in the procedure body traverses the list until it finds the proper position for insertion. It then inserts the argument such that the resulting data structure is again a sorted list. Our analysis, Bohne, is capable of verifying that the postcondition holds at the end of the procedure `insert`, that data structure invariants are preserved, and that there are no run-time errors such as null pointer dereferences. In order to establish these properties, Bohne derives a complex loop invariant shown in Fig. 3.

The main difficulties for inferring this invariant are: (1) it contains universal quantifiers over an unbounded domain and (2) it contains constructs such as reachability, numerical properties, and uninterpreted function symbols.

Bohne infers universally quantified invariants using symbolic shape analysis based on Boolean heaps [53, 43]. This approach can be viewed as a generalization of predicate abstraction or a symbolic approach to parametric shape analysis. Abstraction predicates can be Boolean-valued state predicates (which are either true or false in a given state, such as `prev_le_n`) or predicates denoting sets of heap objects in a given state (which are true of a *given object* in a *given state*, such as `reach_curr`). The latter serve as building

```
class Node {
  public int data;
  public Node next;
}
class SortedList {
  private static Node first;
  /*: public static specvar content :: objset;
      vardefs "content == {v. v ≠ null ∧ next* first v}";

      invariant "tree [next]";
      invariant "first = null ∨ (∀ n. n.next ≠ first)";
      invariant "∀ v. v ∈ content ∧ v.next ≠ null ⟶ v..Node.data ≤ v.next.data";
      invariant "∀ v. v ≠ null ∧ v.next ≠ null ⟶ v.next ∈ content";
  */
  public static void insert (Node n)
    /*: requires "n ≠ null ∧ n ∉ content"
        modifies content
        ensures "content = old content ∪ {n}" */
  {/*: specvar reach_curr :: objset;
       vardefs "reach_curr == {v. next* curr v}";
       specvar prev_le_n :: bool;
       vardefs "prev.data <= n.data"; */
    Node prev = null;
    Node curr = first;
    while ((curr != null) && (curr.data < n.data)) {
      //: track(reach_curr); track(prev_le_n);
      prev = curr;
      curr = curr.next;
    }
    n.next = curr;
    if (prev != null) prev.next = n;
    else first = n;
  }
}
```

Fig. 2. Insertion into a sorted list

$$
\begin{array}{l}
\text{tree [next]} \wedge (\text{first} = \text{null} \vee (\forall\, n.\, n.\text{next} \neq \text{first})) \wedge \\
(\forall\, v.\, v \in \text{content} \wedge v.\text{next} \neq \text{null} \longrightarrow v.\text{data} <= v.\text{next.data}) \wedge \\
(\forall\, v\ w.\, v \neq \text{null} \,\&\, w \neq \text{null} \,\&\, v.\text{next} = w \longrightarrow w \in \text{content}) \wedge \\
n \wedge \text{null} \wedge n \notin \text{content} \wedge \text{content} = \text{old content} \wedge \\
(\text{curr} \wedge \text{null} \longrightarrow \text{curr} \in \text{content}) \wedge (\text{prev} = \text{null} \longrightarrow \text{first} = \text{curr}) \wedge \\
(\text{prev} \wedge \text{null} \longrightarrow \text{prev} \in \text{content} \wedge \text{prev} \notin \text{reach_curr} \wedge \\
\qquad\qquad\qquad\qquad \text{prev.next} = \text{curr} \wedge \text{prev_le_n})
\end{array}
$$

Fig. 3. Loop invariant for procedure `SortedList.insert`

blocks of the inferred universally quantified invariants. The `track(...)` annotation is used as a hint on which predicates the analysis should use for the abstraction of which code fragments.

To reduce the annotation burden we use a syntactic analysis to infer abstraction predicates automatically. Furthermore, parts of the invariant often literally come from the procedure's precondition. In particular, data structure invariants are often preserved as long as the heap is not mutated. We therefore precede the symbolic shape analysis phase with an analysis that propagates precondition conjuncts across the control-flow graph of the procedure's body. Using this propagation technique we are able to infer the first six conjuncts of the invariant. The symbolic shape analysis phase makes use of this partial invariant to infer the full invariant shown in Fig. 3.

Bohne's symbolic shape analysis enables the combination of different decision procedures. Thereby the inferred invariants communicate information between the individual decision procedures, as illustrated with the following example. Figure 4 shows one of the generated verification conditions for the `insert` procedure. It expresses the fact that the sortedness property is reestablished after executing the path from the exit point of the loop

$$
\begin{array}{l}
\mathrm{I} \wedge \neg(\mathrm{curr.data} < \mathrm{n.data}) \wedge \mathrm{prev} \neq \mathrm{null} \wedge \\
\mathrm{next_1} = \mathrm{next}[\mathrm{n} := \mathrm{curr}][\mathrm{prev} := \mathrm{n}] \wedge \\
\mathrm{content_1} = \{\mathrm{v}.\ \mathrm{v} \neq \mathrm{null} \wedge \mathrm{next_1}^*\ \mathrm{first\ v}\} \wedge \\
\mathrm{v} \in \mathrm{content_1} \wedge \mathrm{v.next_1} \neq \mathrm{null} \longrightarrow \mathrm{v.data} \leq \mathrm{v.next_1.data}
\end{array}
$$

Fig. 4. Verification condition for preservation of sortedness

through the if-branch of the conditional to the procedure's return point. The symbol "I" denotes the loop invariant given in Fig. 3. This verification condition is valid. Its proof requires the fact $\texttt{content}' = \texttt{content} \cup \{\texttt{n}\}$; denote this fact P. P follows from the given assumptions. The MONA decision procedure is able to conclude P by expanding the definitions of the abstract sets content and content′. However, MONA is not able to prove the verification condition, because proving its conclusion requires reasoning over integers. On the other hand, the CVC Lite decision procedure is able to prove the conclusion given the fact P by reasoning over the abstract sets without expanding their definitions, but it is not able to conclude P from the assumptions, because this deduction step requires reasoning over reachability. In order to communicate P between the two decision procedures, Bohne infers, in addition to the loop invariant I, an invariant for the procedure's return point that includes the missing fact P. This invariant enables CVC Lite to prove the verification condition.

3 Context-Sensitive Abstraction

We next describe the symbolic shape analysis algorithm implemented in Bohne. What makes this algorithm unique is the fact that abstract transition relations are computed on-demand in each fixpoint iteration taking into account the *context* that approximates previously explored abstract states. This approach allows the algorithm to take advantage of precomputed abstract transition relations from previous fixpoint iterations, while maintaining sufficient precision for the analysis of linked data structures by recomputing the transitions when the context changes in a significant way.

3.1 Reachability Analysis

The input of Bohne is the procedure to be analyzed, preconditions specifying the initial states of the procedure, and a set of abstraction predicates. Bohne converts the procedure into a set of guarded commands that correspond to the loop-free paths in the control-flow graph.

The pseudo code of Bohne's top-level fixpoint computation loop is shown in Figure 5. The analysis first abstracts the conjunction of the procedure's preconditions obtaining an initial set of abstract states. It then computes an abstract reachability tree. Each node in this tree is labeled by a program location and a set of abstract states, the root being labeled by the initial location and the abstraction of the preconditions. The edges in the tree are labeled by guarded commands. The reachability tree keeps track of abstract traces which are used for the analysis of abstract counterexamples.

For each unprocessed node in the tree, the analysis computes the abstract postcondition for the associated abstract states and all outgoing transitions of the corresponding program location. Transitions are abstracted context-sensitively, taking into account the previously discovered reachable abstract states for the associated program location. Whenever the dif-

```
proc Reach(init : precondition formula,
           ℓ_init : initial program location,
           T : set of guarded commands) =
  let init# = abstract(init)
  let root = ⟨location = ℓ_init; states = init#; sons = ∅⟩
  let unprocessed = {root}
  while unprocessed ≠ ∅ do
    choose n ∈ unprocessed
    for all (n.location, c, ℓ') ∈ T do
      let context = { m.states | m.location = n.location }
      let old = { m.states | m.location = ℓ' }
      let new = AbstractPost(c, context, n.states) − old
      if new ≠ ∅ then
        let n' = ⟨location = ℓ'; states = new; sons = ∅⟩
        n.sons := n.sons ∪ {(c, n')}
        unprocessed := unprocessed ∪ {n'}
    unprocessed := unprocessed − {n}
  return root
```

Fig. 5. Reachability analysis in Bohne

ference between the already discovered abstract states of the post location and the abstract post states of the processed transition is non-empty, a new unprocessed node is added to the tree. The analysis stops after the list of unprocessed nodes becomes empty, indicating that the fixpoint is reached. After termination of the reachability analysis, Bohne annotates the original procedure with the computed loop invariants and passes the result to the verification condition generator.

Focusing on algorithmic aspects, we next give a description of the abstract domain, abstraction function, and the abstract post operator.

3.2 Symbolic Shape Analysis

Following the framework of abstract interpretation [11], a static analysis is defined by lattice-theoretic domains and by fixpoint iteration over the domains. Symbolic shape analysis can be seen as a generalization of predicate abstraction [22]. For *predicate abstraction* the analysis computes an invariant; the fixpoint operator is an abstraction of the *post* operator; the concrete domain consists of sets of states (represented by closed formulas), and the abstract domain of a finite lattice of closed formulas.

Abstract Domain. Let Pred be a finite set of abstraction predicates $p(v)$ with an implicit free variable v ranging over heap objects. A *cube* C is a partial mapping from Pred to $\{0,1\}$. We call a total cube *complete*. We say that predicate p occurs positively (occurs negatively, does not occur) in C if $C(p) = 1$ ($C(p) = 0$, $C(p)$ is undefined). We denote by Cubes the set of all cubes. An abstract state is a subset of cubes, which we call a *Boolean heap*. The abstract domain is given by sets of Boolean heaps, i.e. sets of sets of cubes: $\mathsf{AbsDom} = 2^{2^{\mathsf{Cubes}}}$.

Meaning Function. The meaning function γ is defined on cubes, Boolean heaps, and sets

of Boolean heaps as follows:

$$\gamma(C) = \bigwedge_{p \in \mathsf{Pred} \cap \mathsf{dom}(C)} p^{C(p)}, \quad \gamma(H) = \forall v.\ \bigvee_{C \in H} \gamma(C), \quad \gamma(\mathcal{H}) = \bigvee_{H \in \mathcal{H}} \gamma(H)$$

where $p^1 = p$ and $p^0 = \neg p$

The meaning of a cube C is the conjunction of the properly signed predicates in Pred. A Boolean heap H describes all concrete states whose heap is partitioned according to the cubes in H. The meaning of a set $\mathcal{H}$ of Boolean heaps is the disjunction of the meaning of all its elements.

Lattice Structure. Define a partial order $\sqsubseteq$ on cubes by:

$$C \sqsubseteq C' \stackrel{def}{\iff} \forall p \in \mathsf{Pred}.\ C'(p) = C(p) \ \vee\ (C'(p) \text{ is undefined}).$$

For a cube C and Boolean heap H we write $C \in_c H$ as a short notation for the fact that C is complete and there exists $C' \in H$ such that $C \sqsubseteq C'$. The partial order $\sqsubseteq$ is extended from cubes to a preorder on Boolean heaps:

$$H \sqsubseteq H' \stackrel{def}{\iff} \forall C \in H.\ \exists C' \in H'.\ C \sqsubseteq C'.$$

For notational convenience we identify Boolean heaps up to subsumption of cubes, i.e. up to equivalence under the relation $(\sqsubseteq \cap \sqsubseteq^{-1})$. We then identify $\sqsubseteq$ with the partial order on the corresponding quotient of Boolean heaps. In the same way we extend $\sqsubseteq$ from Boolean heaps to a partial order on the abstract domain. These partial orders induce Boolean algebra structures. We denote by $\sqcap$, $\sqcup$ and $\dot{\neg}$ the meet, join and complement operations of these Boolean algebras. Bohne usees binary decision diagrams (BDDs) [9] to implement Boolean heaps, the abstract domain, and operations of the Boolean algebras.

Context-sensitive Cartesian post. The abstract post operator implemented in Bohne is a refinement of the abstract post operator presented in [43]. Its core is given by the *Cartesian post operator*. This operator maps a guarded command c, and a set of Boolean heaps $\mathcal{H}$ to a set of Boolean heaps as follows:

$$\begin{array}{l}\mathsf{CartesianPost}(c, \mathcal{H}) = \\ \quad \textbf{let } \mathsf{cpost}(c, C) = \bigsqcap \{\, C' \mid \forall p \in \mathsf{Pred}.\ C \sqsubseteq \mathsf{wlp}^{\#}(c, p^{C'(p)}) \,\} \\ \quad \textbf{in } \{\, \{\, \mathsf{cpost}(c, C) \mid C \in_c H \,\} \mid H \in \mathcal{H} \,\}.\end{array}$$

The actual abstraction occurs in the computation of $\mathsf{wlp}^{\#}(c, F)$ which is defined by:

$$\mathsf{wlp}^{\#}(c, F) = \{\, C \mid \gamma(C) \models \mathsf{wlp}(c, F) \,\}\ .$$

The Cartesian post maps each Boolean heap H in $\mathcal{H}$ to a new Boolean heap H'. For a given state s satisfying $\gamma(H)$, a cube C in H represents a set of heap objects in s. The Cartesian post computes the local effect of command c on each set of objects which is represented by some complete cube in H: each complete cube C in H is mapped to the smallest cube $\mathsf{cpost}(c, C)$ that represents at least the same set of objects in the post states under command c. Consequently, each object in a given post state is represented by some

cube in the resulting Boolean heap H', i.e. all post states satisfy $\gamma(H')$. The effect of c on the objects represented by some cube is expressed in terms of weakest preconditions wlp of abstraction predicates. These are abstracted by $\mathsf{wlp}^{\#}$.

Computing the effect of c for each cube in H locally implies that we do not take into account the full information provided by H. This becomes an inherent problem if updated predicates express non-local properties such as reachability. As an example, consider a Boolean heap H that contains two cubes

$$C_1 = [(x = v) \mapsto 1, (y = v) \mapsto 0, (\mathsf{next}^* z\ v) \mapsto 1] \text{ and}$$
$$C_2 = [(x = v) \mapsto 0, (y = v) \mapsto 1, (\mathsf{next}^* z\ v) \mapsto 0]\ .$$

Cube C_1 describes an object which is pointed to by a stack variable x and reachable from some other stack variable z following field next. Cube C_2 describes a second object which is pointed to by stack variable y, but which is not reachable from z. If we consider a field update $(c = (x.\mathsf{next} := y))$ then after the update y is reachable from z. However we have

$$\mathsf{cpost}(c, C_2) \not\sqsubseteq [(\mathsf{next}^* z\ v) \mapsto 1]$$

because C_2 is updated independently of C_1. In principle one can strengthen the abstraction of weakest preconditions by taking into account the Boolean heap H for which the post is computed. In fact we have

$$\gamma(H) \wedge (y = v) \models \mathsf{wlp}(c, \mathsf{next}^* z\ v)\ .$$

This strengthening would result in a more precise Cartesian post, but as a consequence abstract weakest preconditions would have to be recomputed for each Boolean heap to which the Cartesian post is applied. This would make the analysis infeasible. Nevertheless, such global context information is valuable when updated predicates describe global properties such as reachability. Therefore, we would like to strengthen the abstraction using some global information, accepting that abstract weakest preconditions have to be recomputed occasionally. We introduce the *context-sensitive Cartesian post* to allow this kind of strengthening:

$$\begin{array}{l} \mathsf{CSCartesianPost}(c, \Gamma, \mathcal{H}) = \\ \textbf{let}\ \mathsf{cpost}(c, C) = \bigsqcap \{\, C' \mid \forall p \in \mathsf{Pred}.\ C \sqsubseteq \mathsf{wlp}^{\#}(c, \Gamma, p^{C'(p)}) \,\} \\ \textbf{in}\ \{\, \{\, \mathsf{cpost}(c, C) \mid C \in_c H \,\} \mid H \in \mathcal{H} \,\} \end{array}$$

$$\text{where} \quad \mathsf{wlp}^{\#}(c, \Gamma, F) = \{\, C \mid \Gamma \wedge \gamma(C) \models \mathsf{wlp}(c, F) \,\} \qquad (1)$$

The formula Γ is the key tuning parameter that controls the tradeoff between precision and efficiency of the analysis. We impose a restriction on Γ: we say that Γ is a *context formula* for a set of Boolean heaps $\mathcal{H}$ if $\gamma(\mathcal{H})$ implies Γ. In order to ensure soundness, we require that for all applications $\mathsf{CSCartesianPost}(c, \Gamma, \mathcal{H})$ of the context-sensitive Cartesian post Γ is a context formula for $\mathcal{H}$.

Figure 6 gives an implementation of the context-sensitive Cartesian post operator that exploits the representation of Boolean heaps as BDDs. First it precomputes an abstract transition relation $c^{\#}$ which is expressed in terms of cubes over primed and unprimed abstraction predicates. After that it computes the relational product of $c^{\#}$ and each Boolean heap. The relational product conjoins a Boolean heap with the abstract transition relation,

proc CSCartesianPost(c, Γ : context formula, $\mathcal{H}$: AbsDom) : AbsDom =
  let $c^\#$ = Cubes
  if $c^\#$ is precomputed for (c, Γ) **then** $c^\# := \text{lookup}(c, \Gamma)$
  else foreach $p \in$ Pred **do**
    $c^\# := c^\# \sqcap \left(\begin{array}{l} [p' \mapsto 1] \sqcap \overline{\mathsf{wlp}^\#(c, \Gamma, \neg p)} \sqcup \\ [p' \mapsto 0] \sqcap \overline{\mathsf{wlp}^\#(c, \Gamma, p)} \end{array} \right)$
  let $\mathcal{H}' = \emptyset$
  foreach $H \in \mathcal{H}$ **do**
    let $H' = \mathsf{RelationalProduct}(H, c^\#)$
    $\mathcal{H}' := \mathcal{H}' \sqcup \{H'\}$
  return $\mathcal{H}'$

Fig. 6. Context-sensitive Cartesian post

projects the unprimed predicates, and renames primed to unprimed predicates in the resulting Boolean heap. Note that the abstract transition relation only depends on command c and the context formula Γ. This allows us to cache abstract transition relations and avoid their recomputation in later fixpoint iterations if Γ is unchanged.

Splitting. The Cartesian post operator maps each Boolean heap in a set of Boolean heaps to one Boolean heap. This means that in terms of precision the Cartesian post does not exploit the fact that the abstract domain is given by *sets* of Boolean heaps. In the following we describe an operation that splits a Boolean heap into a set of Boolean heaps. It is similar to the *focus* operation in TVLA [47]. Splitting maintains important invariants of Boolean heaps that result from best abstractions of concrete states. We split Boolean heaps before applying the Cartesian post. This increases the precision of the analysis by carefully exploiting that the abstract domain is disjunctive complete.

Traditional shape analyses precisely keep track of objects which are pointed to by stack variables. This information is crucial for a precise analysis. In order to keep track of these objects we use abstraction predicates of the form $(x = v)$ where x is some stack variable. Since these predicates denote singleton sets, i.e. each of them is true for exactly one object on the heap, we call them *singleton predicates*. Consequently, if a Boolean heap H is the the result of applying the best abstraction with respect to γ to some concrete state then for every singleton predicate p it contains exactly one complete cube with a positive occurrence of p. Boolean heaps resulting from the Cartesian post might not have this property. This makes the analysis imprecise. Therefore we split each Boolean heap before application of the Cartesian post into a set of Boolean heaps, such that the above property is reestablished. Let P be the subset of abstraction predicates denoting singletons then the *splitting operator* is defined as follows:

$$\begin{aligned} \mathsf{Split}(\mathcal{H}) &= \mathsf{split}(P, \mathcal{H}) \\ \mathsf{split}(\emptyset, \mathcal{H}) &= \mathcal{H} \\ \mathsf{split}(\{p\} \cup P', \mathcal{H}) &= \textbf{let } C_p = [p \mapsto 1] \textbf{ and } C_{\neg p} = [p \mapsto 0] \textbf{ in} \\ &\quad \textstyle\bigcup_{H \in \mathcal{H}} \mathsf{split}(P', \{\, H \sqcap \{C_{\neg p}\} \sqcup \{C\} \mid C \in_c (H \sqcap \{C_p\}) \,\}). \end{aligned}$$

The splitting operator takes a set of Boolean heaps $\mathcal{H}$ as arguments. For each singleton predicate p and Boolean heap H it splits H into a set of Boolean heaps. Each of the

$$\mathsf{abstract}(F) = \textbf{let } H = \overline{\{\, C \mid C \models \neg F \,\}} \textbf{ in } \mathsf{Clean}(F, \mathsf{Split}(H))$$

$$\begin{array}{l}\textbf{proc } \mathsf{AbstractPost}(c,\ \mathsf{context} : \mathsf{AbsDom},\ \mathcal{H}_0 : \mathsf{AbsDom}) : \mathsf{AbsDom} = \\ \quad \textbf{let } \mathcal{H} = \mathsf{Clean}(\mathsf{guard}(c), \mathsf{Split}(\mathcal{H}_0)) \\ \quad \textbf{let } \Gamma = \kappa(\mathsf{context} \sqcup \mathcal{H}) \\ \quad \textbf{return } \mathsf{CSCartesianPost}(c, \Gamma, \mathcal{H})\end{array}$$

Fig. 7. Bohne's abstract post operator

resulting Boolean heaps corresponds to H, but contains only one of the complete cubes in H that have a positive occurrence of p. The splitting operator is sound, i.e. satisfies: $\gamma(\mathsf{Split}(\mathcal{H})) = \gamma(\mathcal{H})$.

Cleaning. Splitting might introduce unsatisfiable Boolean heaps, because it is done propositionally without taking into account the semantics of predicates. Unsatisfiable Boolean heaps potentially lead to spurious counterexamples and hence should be eliminated. The same applies to cubes that are unsatisfiable with respect to other cubes within one Boolean heap. We use a *cleaning operator* [1] to eliminate unsatisfiable Boolean heaps and unsatisfiable cubes within satisfiable Boolean heaps. At the same time we strengthen the Boolean heaps with the guard of the commands before the actual computation of the Cartesian post. The cleaning operator is defined as follows:

$$\begin{array}{l}\mathsf{Clean}(F, \mathcal{H}) = \textbf{let } \mathcal{H}_1 = \{\, H \in \mathcal{H} \mid F \wedge \gamma(H) \not\models \mathsf{false} \,\} \textbf{ in} \\ \qquad\qquad \{\, \{\, C \in_c H \mid F \wedge \gamma(H) \wedge \gamma(C) \not\models \mathsf{false} \,\} \mid H \in \mathcal{H}_1 \,\}.\end{array}$$

The operator Clean takes as arguments a formula F (e.g. the guard of a command) and a set of Boolean heaps. It first removes all Boolean heaps that are unsatisfiable with respect to F. After that it removes from each remaining Boolean heap H all complete cubes which are unsatisfiable with respect to F and H. The cleaning operator is sound, i.e. strengthens $\mathcal{H}$ with respect to F:

$$F \wedge \gamma(\mathcal{H}) \models \gamma(\mathsf{Clean}(F, \mathcal{H})) \models \gamma(\mathcal{H}) \ .$$

Obviously the cleaning and splitting operators bear the danger of an exponential blowup. This can be avoided, e.g. by giving up precision and enforcing a polynomial bound by only considering cubes up to a fixed length. However, in practice this does not seem to be necessary, because Boolean heaps are relatively sparse and contain only few complete cubes.

Abstract post operator. Figure 7 defines the abstract post operator used in Bohne. It is defined as the composition of the splitting, cleaning, and the Cartesian post operator. The function κ is a *context operator*. A context operator is a monotone mapping from sets of Boolean heaps to a context formula. It controls the trade-off between precision and efficiency of the abstract post operator. Our choice of κ is described in the next section. Figure 7 also defines the abstraction function that is used to compute the initial set of Boolean heaps. For abstracting a formula F the function abstract first computes a Boolean heap H which is the complement of an under-approximation of $\neg F$. It then splits H with

[1] The cleaning operator resembles the *coerce* operation in TVLA [47].

$$
\begin{array}{l}
\mathsf{Var} - \text{object-valued program variables} \\
\mathsf{instantiate}(H : \text{Boolean heap}) : \text{formula} = \\
\quad \textbf{let } \mathsf{cube}(x) = \bigsqcup(H \sqcap \{[(x = v) \mapsto 1]\}) \textbf{ in} \\
\quad \bigwedge_{x \in \mathsf{Var}} \gamma(\mathsf{cube}(x))[v := x] \\
\kappa(\mathcal{H}) = \textbf{let } H = \bigsqcup \mathcal{H} \textbf{ in } \mathsf{instantiate}(H)
\end{array}
$$

Fig. 8. Context instantiation and the context operator κ

respect to singleton predicates and strengthens the result by the original formula F. We compute the abstraction indirectly because it allows us to reuse all the functionality that we need for computing the abstract post operator. We also avoid computing the best abstraction function for the abstract domain, because the computational overhead is not justified in terms of the gained precision.

Assuming that κ is in fact a context operator, soundness of AbstractPost follows from the soundness of all its component operators. Note that soundness is still guaranteed if the underlying validity checker is incomplete.

4 Context Instantiation

The context information used to strengthen the abstraction is given by the set of Boolean heaps that are already discovered at the respective program location. If we take into account all available context for the abstraction of a transition then we need to recompute the abstract transition relation in every iteration of the fixed point computation. Otherwise the analysis would be unsound. In order to avoid unnecessary recomputations we use the operator κ to abstract the context by a context formula that less likely changes from one iteration to the next. For this purpose we introduce a domain-specific quantifier instantiation technique. We use this technique not only in connection with the context operator, but more generally to eliminate any universal quantifier in a decision procedure query that originates from the concretization of a Boolean heap. This eliminates the need for the underlying decision procedures to deal with quantifiers.

We observed that the most valuable part of the context is the information available over objects pointed to by program variables. This is due to the fact that transitions always change the heap with respect to these objects. We therefore instantiate Boolean heaps to objects pointed to by stack variables. Bohne automatically adds an abstraction predicate of the form $(x = v)$ for every object-valued program variable x. A syntactic backwards analysis of the procedure's assert statements and postcondition is used to determine which of these predicates are relevant at each program point.

Figure 8 defines the function instantiate. It uses the above mentioned predicates to instantiate a Boolean heap H to a quantifier free formula (assuming predicates itself are quantifier free). For every program variable x it computes the least upper bound of all cubes in H which have a positive occurrence of predicate $(x = v)$. The resulting cube is concretized and the free variable v is substituted by program variable x. The function κ maps a set of Boolean heaps $\mathcal{H}$ to a formula by taking the join of $\mathcal{H}$ and instantiating

the resulting Boolean heap. One can show that κ is indeed a context operator, i.e. κ is monotone and the resulting formula is a context formula for $\mathcal{H}$.

5 Semantic Caching

Abstracting context does not avoid that abstract transition relations have to be recomputed occasionally in later fixpoint iterations. Whenever we recompute abstract transition relations we would like to reuse the results from previous abstractions. We do this on the level of decision procedure calls by caching the queries and the results of the calls. Syntactic caching of decision procedure queries has been used before (e.g. [2] mentions its use in the SLAM system [3]). The problem with simple syntactic caching of formulas in shape analysis is that the context formulae are passed to the decision procedure as part of the queries, so a simple syntactic approach is ineffective. However, the context consists of all discovered abstract states at the current iteration. Therefore, the context changes monotonically from one iteration to the next. The monotonicity of the context operator κ guarantees that context formulae, too, increase monotonically with respect to the entailment order. We therefore cache formulas by keeping track of the partial order on the context. Since context formulae occur in the antecedents of the queries, this allows us to reuse negative results of entailment checks from previous fixpoint iterations. This method is effective because in practice the number of entailments which are invalid exceeds the number of valid ones.

Furthermore, formulas are cached up to alpha equivalence. Since the cache is self-contained, this enables caching results of decision procedure calls not only across different fixpoint iterations for one procedure, but even across the analysis of different procedures. This yields substantial improvements for procedures that exhibit some similarity, which opens up the possibility of using our analysis in an interactive context. For example, we verified a procedure inserting an element into a sorted list (see `SortedList.add` in Figure 9) and repeated the analysis without erasing the cache on a modified version of the same procedure where two commuting assignments were exchanged. About 90% of the results to decision procedure calls were found in the cache, causing that running time went down from 11s to 3s.

6 Propagation of Precondition Conjuncts

It often happens that parts of loop invariants literally come from the procedure's preconditions. A common situation where this occurs is that a procedure executes a loop to traverse a data structure performing only updates on stack variables and after termination of the loop the data structure is manipulated. In such a case the data structure invariants are trivially preserved while executing the loop. Using an expansive symbolic shape analysis to infer such invariants is inappropriate. We therefore developed a fast but effective analysis that propagates conjuncts from the precondition across the procedure's control-flow graph. This propagation precedes the symbolic shape analysis, such that the latter is able to assume the previously inferred invariants.

The propagation analysis works as follows: it first splits the procedure's precondition into a conjunction of formulas and assumes all conjuncts at all program locations. It then recursively removes a conjunct F at program locations that have an incoming control flow edge from some location where either (1) F has been previously removed or (2) where F

benchmark	used DP	# predicates total (manually supplied)	# validity checker calls total (cache hits)	running time total (DP)
DLL.addLast	MONA	7 (0)	118 (19%)	2s (69%)
List.reverse	MONA	7 (2)	371 (22%)	4s (72%)
SortedList.add	MONA, CVC lite	16 (1)	368 (40%)	11s (65%)
Skiplist.add	MONA	20 (0)	787 (44%)	26s (57%)
Tree.add	MONA	13 (0)	358 (31%)	31s (92%)
ParentTree.add	MONA	13 (0)	362 (32%)	33s (91%)
Linear.arrayInv	CVC lite	7 (5)	882 (52%)	57s (97%)

Fig. 9. Results of Experiments

is not preserved under post of the associated command. After termination of the analysis (none of the rules for removal applies anymore) the remaining conjuncts are guaranteed to be invariants at the corresponding program points.

The preservation of conjuncts is checked by discharging a verification condition (via decision procedure calls). The use of decision procedures makes this analysis more general than the syntactic approach for computing frame conditions for loops used in ESC/Java-like desugaring of loops [15]. In particular, the propagation is still applicable in the presence of heap manipulations that preserve the invariants in each loop-free code fragment. Unlike the Houdini tool [13], precondition conjunct propagation does not attempt to invent new predicates.

7 Experiments

We applied Bohne to verify operations on various data structures. Our experiments cover data structures such as singly-linked lists, doubly-linked lists, two-level skip lists, trees, trees with parent pointers, sorted lists, and arrays. The verified properties include: (1) absence of run-time errors, such as null pointer dereferences and array bound violations; (2) complex data structure consistency properties, such as preservation of the tree structure, array invariants, as well as sortedness; and (3) procedure contracts, stating e.g. how the set of elements stored in a data structure is affected by the procedure.

Figure 9 shows the results for a collection of benchmarks running on a 2 GHz Pentium M with 1 GB memory. The Jahob system is implemented in Objective Caml and compiled to native code. Running times include inference of loop invariants. This time dominates the time for a final check (using verification-condition generator) that the resulting loop invariants are sufficient to prove the postcondition. The benchmarks can be found on the Jahob project web page [27]. The version of Bohne used to generate these results uses a simple analysis of the source code to determine most of the abstraction predicates automatically; the number of predicates in parentheses indicates the additional predicates that we needed to specify to make the symbolic shape analysis sufficiently precise. Note that we did not need to specify how these predicates change in response to program statements; this is computed automatically by Bohne. Note also that our examples are not stand-alone programs that build and then traverse their own data structures. Instead, our examples use assume-guarantee reasoning of Jahob to verify procedures with non-trivial preconditions, postconditions and representation invariants. As a result, these examples can be used in the context of larger programs that are verified by more scalable analysis, as demonstrated in the Hob project [33].

benchmark	DLL.addLast	SortedList.add	Skiplist.add	Tree.add
	no context ($\Gamma = \mathsf{true}$)			
running time	2s	14s	29s	71s
DP calls (cache hits)	118 (20%)	457 (32%)	1110 (51%)	1024 (51%)
	context-sensitive without instantiation ($\kappa = \mathsf{id}$)			
running time	4s	24s	72s	473s
DP calls (cache hits)	178 (23%)	445 (22%)	1031 (38%)	742 (13%)
	context-sensitive with instantiation			
running time	2s	11s	26s	32s
DP calls (cache hits)	118 (19%)	368 (40%)	787 (44%)	358 (31%)

Fig. 10. Effect of context-sensitive abstraction and context instantiation

We also examined the impact of context-sensitive abstraction and context instantiation on the running time of the analysis. The results are shown in Table 10. As expected, running times for context-sensitive abstraction with instantiation disabled are significantly higher (2-8 times) than with instantiation enabled. Without context instantiation abstract transition relations have to be recomputed many times and caching of decision procedure calls is less effective. If context-sensitive abstraction is disabled completely the analysis not only becomes less precise (e.g. the analysis failed to verify the SortedList and SkipList examples without context) but also in many cases slower. Most likely the less precise analysis needs to explore a larger part of the abstract state space.

Note that our implementation of the algorithm is not highly tuned in terms of aspects orthogonal to Bohne's algorithm, such as type inference for internally manipulated Isabelle formulas. We expect that the running times would be notably improved using more efficient implementation of Hindley-Milner type reconstruction. In previous benchmarks without type reconstruction in average 97% of the time was spent in the decision procedures. The most promising directions for improving the analysis performance are therefore 1) deploying more efficient decision procedures, and 2) further reducing the number of decision procedure calls.

In addition to the presented examples, we have used the verification condition generator to verify examples such as array-based implementations of containers and implementations of association lists. Bohne can also infer loop invariants in such examples.

8 Conclusions

We described Bohne, a data structure verification algorithm based on symbolic shape analysis that infers invariants about sets given by predicates on objects. We showed how to fruitfully combine this abstraction with a collection of decision procedures that operate on independent subgoals of the same proof obligation. We deployed a range of techniques that improve the running time of the analysis and the level of automation compared to direct application of the algorithm. These techniques include context-dependent abstraction, semantic caching of formulas, propagation of conjuncts, and domain-specific quantifier instantiation. Our experience with the Bohne analysis in the context of the Hob and Jahob data structure verification systems suggests that it is effective for verifying a wide range of data structures with user-defined procedure contracts. The verified properties go beyond traditional shape properties such as treeness and include the characterization of data structure operations in terms of changes to their content.

References

[1] I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*, 2005.

[2] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Tool Paper, CAV*, 2004.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[4] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16^{th} Int. Conf. on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518, 2004.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[6] D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In T. Ball and R. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006, Seattle, WA, August 16-20)*, LNCS 4144, pages 532–546. Springer-Verlag, Berlin, 2006.

[7] J. Bingham and Z. Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs. Technical Report TR-2005-19, UBC Department of Computer Science, September 2005.

[8] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard. On using first-order theorem provers in a data structure verification system. Technical Report MIT-CSAIL-TR-2006-072, MIT, November 2006. http://hdl.handle.net/1721.1/34874.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[10] D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning in expressive description logics with fixpoints based on automata on infinite trees. In *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI'99)*, pages 84–89, 1999.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, 1977.

[12] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS'06*, 2006.

[13] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001. Springer-Verlag.

[14] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. 29th ACM POPL*, 2002.

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

[16] P. Fradet and D. L. Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.

[17] L. Georgieva and P. Maier. Description logics for shape analysis. In *Proc. 3rd SEFM*, pages 321–330, 2005.

[18] R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.

[19] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proc. 8th Workshop on Languages and Compilers for Parallel Computing*, 1995.

[20] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, pages 240–260, 2006.

[21] E. Grädel. Decision procedures for guarded logics. In *Automated Deduction - CADE16. Proceedings of 16th International Conference on Automated Deduction, Trento, 1999*, volume 1632 of *LNCS*. Springer-Verlag, 1999.

[22] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th CAV*, pages 72–83, 1997.

[23] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[24] N. D. Jones and S. S. Muchnick. Chapter 4: Flow analysis and optimization of LISP-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.

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

[26] N. Klarlund and M. I. Schwartzbach. Graph types. In *Proc. 20th ACM POPL*, Charleston, SC, 1993.

[27] V. Kuncak. The Jahob project web page, http://www.mit.edu/∼vkuncak/projects/jahob/, 2006.

[28] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

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

[30] V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006. http://dx.doi.org/10.1007/s10817-006-9042-1.

[31] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV'04*, 2004.

[32] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, 2006.

[33] 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.

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

[35] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, 2005.

[36] T. Lev-Ami. TVLA: A framework for kleene based logic static analyses. Master's thesis, Tel-Aviv University, Israel, 2000.

[37] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE-20*, 2005.

[38] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. Software Testing and Analysis*, 2000.

[39] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In R. Cousot, editor, *Proceedings of the 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*, volume 3148 of *LNCS*, pages 181–198. Springer, 2005.

[40] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005.

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

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

[43] A. Podelski and T. Wies. Boolean heaps. In *Proc. Int. Static Analysis Symposium*, 2005.

[44] S. Ranise and C. G. Zarba. A decidable logic for pointer programs manipulating linked lists, 2005. http://cs.unm.edu/∼zarba/papers/pointers.ps.

[45] J. Reineke. Shape analysis of sets. Master's thesis, Universität des Saarlandes, Germany, June 2005.

[46] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *Proc. 12th ESOP*, 2003.

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

[48] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[49] G. Sutcliffe and C. B. Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[50] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, August 1968.

[51] A. Voronkov. The anatomy of Vampire (implementing bottom-up procedures with code trees). *Journal of Automated Reasoning*, 15(2):237–265, 1995.

[52] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

[53] T. Wies. Symbolic shape analysis. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, September 2004.

[54] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006.

[55] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2006)*, 2006.

[56] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th TACAS*, 2004.

[57] G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *TOCL*, 8(1), 2007.

[58] G. Yorsh, A. Skidanov, T. Reps, and M. Sagiv. Automatic assume/guarantee reasoning for heap-manupilating programs. In *1st AIOOL Workshop*, 2005.