On Verifying Complex Properties using Symbolic Shape Analysis

Thomas Wies   Viktor Kuncak
Karen Zee   Andreas Podelski
Martin Rinard

MPI–I–2006–2–1   April 2006

Authors' Addresses

Thomas Wies
Max-Planck-Institut für Informatik
Saarbrücken, Germany

Viktor Kuncak
MIT Computer Science and Artificial Intelligence Lab
Cambridge, USA

Karen Zee
MIT Computer Science and Artificial Intelligence Lab
Cambridge, USA

Andreas Podelski
Max-Planck-Institut für Informatik
Saarbrücken, Germany

Martin Rinard
MIT Computer Science and Artificial Intelligence Lab
Cambridge, USA

Abstract

One of the main challenges in the verification of software systems is the analysis of 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), and an 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 as well as techniques that Bohne uses to reduce the ammount of annotations and the running time of the analysis.

Contents

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 [39] as well as predicate abstraction [2, 17] and its generalizations [9, 24].

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 programs 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 [35, 42] for linked data structures and uses the key idea of shape analysis: 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.

We next put the core Bohne algorithm in the context of predicate abstraction and parametric shape analysis approaches.

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 [19], and combines it with uninterpreted function symbols in a way that preserves completeness in important cases [43]. 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 [9], indexed predicates [24] 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 [24] 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 [31, 42].

Shape analysis. Shape analyses are precise analyses for linked data structures. They were originally used for compiler optimizations [13,14,18] and lacked precision needed to establish invariants that Bohne is analyzing. Precise data structure analysis for the purpose of verification include [11, 20, 23, 28, 32, 39] and have recently also been applied to verify set implementations [37]. 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 [35, 43, 45–47] or finite differencing [38]. We are currently working on an effort to compare such different analysis on a joint set of benchmarks [22]. Our approach differs from [25] 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 [3, 19, 34]. Shape analysis approaches have also been used to verify sortedness properties [30] relying on manually abstracting sortedness relation.

Recently there has been a resurgence of decision procedures and analyses for linked list data structures [1, 4, 8, 31, 36], where the emphasis is on predictability (decision procedures for well-defined classes of properties of linked lists), 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 decision procedures when they are applicable and using more general reasoning otherwise.

Bohne could also take advantage of logics for reasoning about reachability, such as the logic of reachable shapes [44]. Existing logics, such as guarded fixpoint logic [15] and description logics with reachability [6, 12] are attractive because of their expressive power, but so far no decision procedures for these logics have been implemented. Automated theorem provers such as Vampire [40] and SPASS [41] can be used to reason about properties of linked data structures, but axiomatizing reachability in first-order logic is non-trivial in practice [29, 33] and not possible in general.

1.1 Contributions

We have previously described the general idea of symbolic shape analysis [35] as well as the field constraint analysis decision procedure for combining reachability reasoning with uninterpreted function symbols [43]. In [48] we have described splitting of proof obligations in the context of verifying proof obligations using the Isabelle interactive theorem prover. One of the insights in this paper is that such splitting can be an effective way of combining different reasoning procedures during fixpoint computation in abstract interpretation. These previous techniques are therefore the starting point of this paper. The main contributions of this paper are the following:

1. We introduce a technique for combining different decision procedures through 1) a static analysis that synthesizes Boolean algebra expressions over sets defined by arbitrary abstraction predicates, 2) a proof obligation splitting approach that discharges different conjuncts using different decision procedures, and 3) a verification-condition generator that preserves abstract variables. This approach addresses a key question in extending a Nelson-Oppen style combination to theories that share *sets of elements*. In general, such combination would require guessing and propagating an exponential

number of Boolean algebra expressions. In our approach, symbolic shape analysis [35] synthesizes Boolean algebra expressions that are used as assumptions in decision procedures calls and are therefore shared by all participating decision procedures.

2. 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.

3. 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.

4. 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.

5. We present a domain-specific quantifier instantiation technique that 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, and sorted lists. 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 Bohne and used to communicate information between different decision procedures.

Bohne as component of Hob and Jahob. Bohne is part of the data structure verification frameworks Hob [26, 27] and Jahob [21]. 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

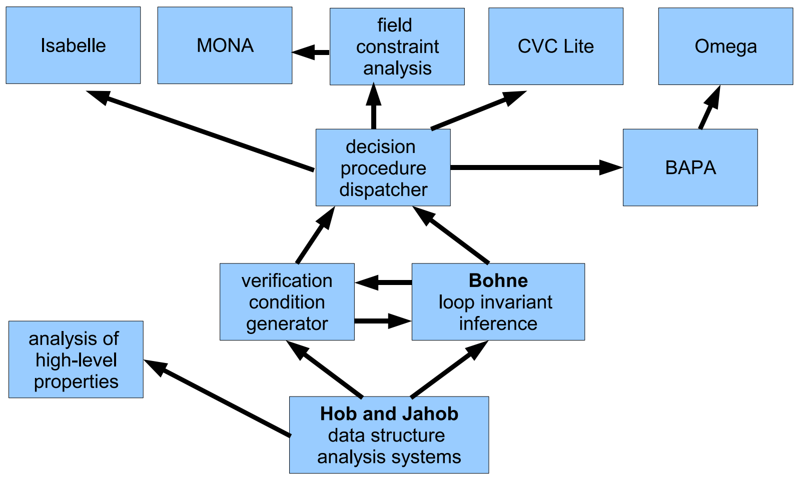
Isabelle
MONA
field constraint analysis
CVC Lite
Omega
decision procedure dispatcher
BAPA
verification condition generator
Bohne loop invariant inference
analysis of high-level properties
Hob and Jahob data structure analysis systems

Figure 1.1: Architecture of the Hob and Jahob Data Structure Analysis Systems

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 1.1 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 a validity checker that combines special purpose decision procedures, a general purpose theorem prover, and reasoning techniques such as field constraint analysis [43].

2 Motivating Example

We illustrate our technique on the procedure `SortedList.insert` shown in Figure 2.1. 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 [34]. 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 field `Node.next`. The construct `rtrancl_pt` is a higher-order function that maps a binary predicate to its reflexive transitive closure. The data structure invariants are specified by the annotation `invariant "..."`. For instance, the first invariant expresses the fact that the field `Node.next` forms trees in the heap, i.e. that `Node.next` is acyclic and injective; the second invariant expresses the fact that the elements stored in the list are sorted in increasing order according to field `Node.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 content of the list is unchanged except for the argument being added.

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. 2.2. The main difficulties for inferring this invariant are: (1) it contains universal quantifiers over an unbounded domain and (2) it requires reasoning over multiple theories, here reasoning over reachability, reasoning over numerical domains, and reasoning over uninterpreted function symbols.

Bohne infers universally quantified invariants using symbolic shape analysis based on Boolean heaps [35,42]. This approach can be viewed as a generalization of predicate abstraction or a symbolic approach to parameteric shape analysis.

```
public final class Node {
  public int data;
  public Node next;
}

public class SortedList {
  private static Node first;
  /*:
    public static specvar content :: objset;
    vardefs "content ==
      {v. v ~= null & rtrancl_pt (% x y. x..Node.next = y) first v}";

    invariant "tree [Node.next]";
    invariant "first = null | (ALL n. n..Node.next ~= first)";
    invariant "ALL v. v : content & v..Node.next ~= null -->
                 v..Node.data <= v..Node.next..Node.data";
    invariant "ALL v w. v ~= null & w ~= null & v..Node.next = w -->
                 w : content";
  */
  public static void insert(Node n)
    /*:
      requires "n ~= null & n ~: content"
      modifies content
      ensures "content = old content Un {n}"
    */
  {
    /*:
      specvar lt_n :: objset;
      vardefs "lt_n == {v. v..Node.data < n..Node.data}";
      specvar curr_prev :: bool;
      vardefs "curr_prev == (prev..Node.next = curr)";
    */
    Node prev = null;
    Node curr = first;
    while ((curr != null) && (curr.data < n.data)) {
      /*:
        track(curr_prev);
        track(lt_n);
      */
      prev = curr;
      curr = curr.next;
    }
    n.next = curr;
    if (prev != null) prev.next = n;
    else first = n;
  }
}
```

Figure 2.1: Insertion into a sorted list

```
tree [Node.next] &
(first = null | (ALL n. n..Node.next ~= first)) &
(ALL v. v : content & v..Node.next ~= null -->
  v..Node.data <= v..Node.next..Node.data) &
(ALL v w. v ~= null & w ~= null & v..Node.next = w -->
  w : content) &
n ~= null & n ~: content &
reach_curr = {v. rtrancl_pt (% x y. x..Node.next = y) curr v} &
content = old content &
(curr ~= null --> curr : content) &
(prev = null --> first = curr) &
(prev ~= null -->
  prev : content & prev ~: reach_curr & prev..Node.next = curr) &
(ALL v. v ~: reach_curr & v : content --> v : lt_n)
```

Figure 2.2: Loop invariant for procedure `SortedList.insert`

Abstraction predicates can be Boolean-valued state predicates (which are either true or false in a given state, such as `curr_prev`) or predicates denoting sets of heap objects in a given state (which are true of a *given object* in a *given state*, such as `lt_n`). The latter serve as building 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 (e.g. predicate `reach_curr` in the loop invariant). 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 accross 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. 2.2.

Bohne's symbolic shape analysis enables the combination of decision procedures by connecting the analysis with a proof obligation splitting approach that discharges different conjuncts using different decision procedures, and a verification-condition generator that preserves abstract variables. Thereby the inferred invariants communicate information between different decision procedures. This combination is best illustrated with an example. Figure 2.3 shows one of the generated verification conditions for the procedure `SortedList.insert`. It expresses the fact that the sortedness property is reestablished after executing the path from the exit point of the loop through the if-branch of the conditional to the procedure's return point. The symbol "`I`" denotes the loop invariant given in Fig. 2.2.

```
I & ~(curr..Node.data < n..Node.data) & prev ~= null &
Node.next' = Node.next[n := curr][prev := n] &
content' =
  {v. v ~= null & rtrancl_pt (% x y. x..Node.next' = y) first v} &
v : content' & n..Node.next' ~= null -->
  v..Node.data <= v..Node.next'..Node.data
```

Figure 2.3: Verification condition for preservation of sortedness

This verification condition is valid. Its proof requires the fact

$$\texttt{content'} = \texttt{content}\ \texttt{Un}\ \{\texttt{n}\}$$

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 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, symbolic shape analysis infers, in addition to the loop invariant, 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 The Bohne Algorithm

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* in form of already 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.

Figure 3.1 gives the pseudo code of Bohne's top-level fixpoint computation loop. 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 in the spirit of lazy abstraction [17]. 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 on-demand taking into account the already discovered reachable abstract states for the associated program location. Whenever the difference between the already discovered abstract states of the post location and the abstract post states of the processed transition

```
proc Reach(init : precondition formula,
           $\ell_{\mathsf{init}}$ : initial program location,
           $T$ : set of guarded commands) $=$
  let $\mathsf{init}^{\#} = \mathsf{abstract}(\mathsf{init})$
  let $\mathsf{root} = \langle \mathsf{location} = \ell_{\mathsf{init}}; \mathsf{states} = \mathsf{init}^{\#}; \mathsf{sons} = \emptyset \rangle$
  let $\mathsf{unprocessed} = \{\mathsf{root}\}$
  while $\mathsf{unprocessed} \neq \emptyset$ do
    choose $n \in \mathsf{unprocessed}$
    for all $(n.\mathsf{location}, c, \ell') \in T$ do
      let $\mathsf{context} = \{\, m.\mathsf{states} \mid m.\mathsf{location} = \ell \,\}$
      let $\mathsf{old} = \{\, m.\mathsf{states} \mid m.\mathsf{location} = \ell' \,\}$
      let $\mathsf{new} = \mathsf{AbstractPost}(c, \mathsf{context}, n.\mathsf{states}) - \mathsf{old}$
      if $\mathsf{new} \neq \emptyset$ then
        let $n' = \langle \mathsf{location} = \ell'; \mathsf{states} = \mathsf{new}; \mathsf{sons} = \emptyset \rangle$
        $n.\mathsf{sons} := n.\mathsf{sons} \cup \{(c, n')\}$
        $\mathsf{unprocessed} := \mathsf{unprocessed} \cup \{n'\}$
    $\mathsf{unprocessed} := \mathsf{unprocessed} - \{n\}$
  return root
```

Figure 3.1: Reachability analysis in Bohne

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, which verifies that the inferred loop invariants are sufficient to prove the target properties.

The algorithm in Figure 3.1 is parameterized by the abstract domain and its associated operators. An abstract state of the analysis is given by a set of bitvectors over abstraction predicates which we call a Boolean heap. It corresponds to a universally quantified Boolean combination of abstraction predicates. A Boolean heap describes all concrete states whose heap is partitioned according to the bitvectors in the Boolean heap. Focusing on algorithmic details, we now give a detailed description of the abstract domain, abstraction function, and the abstract post operator.

3.2 Symbolic Shape Analysis

Following the framework of abstract interpretation [7], 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 [16]. 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:

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

The meaning of a cube C is the conjunction of the predicates in Pred and their negations. A concrete state is represented by a Boolean heap H if all objects in the heap are represented by some cube 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:

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

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:

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

For notational convenience we identify Boolean heaps up to subsumption of cubes, i.e. up to equivalence under the relation ($\sqsubseteq \cup \sqsubseteq^{-1}$). 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 $\overline{\cdot}$ the meet, join and complement operations of these Boolean algebras. Boolean heaps, the abstract domain, and operations of the Boolean algebras are implemented using BDDs [5].

Context-sensitive Cartesian post. The abstract post operator implemented in Bohne is a refinement of the abstract post operator on Boolean heaps that is presented in [35]. Its core is given by the *context-sensitive Cartesian post operator*. This operator maps a guarded command c, a formula Γ, and a set of Boolean heaps $\mathcal{H}$ to a set of Boolean heaps as follows:

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

The actual abstraction is hidden in the computation of the function $\mathsf{wlp}^{\#}$ which is defined by:

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

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 C' that represents at least the same set of objects in the post states under command c. Consequently all objects in a given post state are 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 of abstraction predicates. These are abstracted by the function $\mathsf{wlp}^{\#}$.

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. In principle one can strengthen the abstraction of weakest preconditions by taking into account the Boolean heap for which the post is computed: $\mathsf{wlp}^{\#}(c,\gamma(H),p)$. The abstract post would be more precise, but as a consequence abstract weakest preconditions would have to be recomputed for each Boolean heap. 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. The formula Γ allows this kind of strengthening. It is the key tuning parameter of the analysis. We impose a restriction on Γ to ensure soundness: we say that Γ

```
**proc** $\mathsf{CartesianPost}(c$ : guarded command,
                    $\Gamma$ : context formula,
                    $\mathcal{H} : \mathsf{AbsDom}) : \mathsf{AbsDom} =$
  **let** $\mathsf{c}^{\#} = \mathsf{Cubes}$
  **if** $c^{\#}$ is precomputed for $(c, \Gamma)$ **then** $c^{\#} := \mathrm{lookup}(c, \Gamma)$
  **else foreach** $p \in \mathsf{Pred}$ **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** $\mathcal{H}'$
```

Figure 3.2: Context-sensitive Cartesian post

is a *context formula* for a set of Boolean heaps $\mathcal{H}$ if $\gamma(\mathcal{H})$ implies Γ. Restricting the Cartesian post to context formulas ensures soundness with respect to the best abstract post operator on sets of Boolean heaps.

Figure 3.2 gives an implementation of the 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, projects the unprimed predicates, and renames primed to unprimed predicates in the resulting Boolean heap. Note that that the abstract transition relation only depends on the abstracted command c and the context formula Γ. This allows us to cache abstract transition relations and avoid their recomputation in later fixpoint iterations where 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. The 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 the disjunctive completeness of the abstract domain.

Traditional shape analysis uses the idea of summary nodes to distinguish abstract objects that represent multiple concrete objects from abstract objects that represent single objects. This information is useful for increasing the precision of the abstract post operator. We can mimic this idea by adding abstraction predicates that denote singleton sets, e.g. by adding predicates expressing properties such as that an object is pointed to by some local variable. If a Boolean heap H is the best abstraction of 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 typically do not have this property which 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:

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

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 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:

$$\gamma(\mathsf{Split}(P, \mathcal{H})) \equiv \gamma(\mathcal{H}) \ .$$

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 in the analysis 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* 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:

$$\begin{array}{rcl} \mathsf{Clean}(F, \mathcal{H}) & = & \textbf{let } \mathcal{H}_1 = \{\, H \in \mathcal{H}_0 \mid F \land \gamma(H) \not\models \mathsf{false} \,\} \textbf{ in} \\ & & \displaystyle\bigsqcup_{H \in \mathcal{H}_1} \{\, C \in_c H \mid F \land \gamma(H) \land \gamma(C) \not\models \mathsf{false} \,\} \ . \end{array}$$

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

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

proc $\mathsf{AbstractPost}(c$ : guarded command,
                    $\mathsf{context} : \mathsf{AbsDom}$,
                    $\mathcal{H}_0 : \mathsf{AbsDom}) : \mathsf{AbsDom} =$
    let $\mathcal{H} = \mathsf{Clean}(\mathsf{guard}(c), \mathsf{Split}(\mathcal{H}_0))$
    let $\Gamma = \kappa(\mathsf{context} \sqcup \mathcal{H})$
    return $\mathsf{CartesianPost}(c, \Gamma, \mathcal{H})$
```

Figure 3.3: Bohne's abstract post operator

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:

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

Abstract post operator. Figure 3.3 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 3.3 also defines the abstraction function that is used to compute the initial set of Boolean heaps. For abstracting a formula F the function $\mathsf{abstract}$ first computes a Boolean heap H which is the complement of an under-approximation of $\neg F$. It then splits H with 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 $\mathsf{AbstractPost}$ follows from the soundness of all its component operators. Note that soundness is still guaranteed if the underlying validity checker is incomplete.

3.3 Quantifier 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

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

Figure 3.4: Quantifier instantiation and the context operator κ

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 postconditions is used to determine which of these predicates are relevant at each program point.

Figure 3.4 defines the function $\mathsf{instantiate}$ that uses the above mentioned predicates to instantiate a Boolean heap H to a quantifier free formula (assuming abstraction 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 as described above. One can shown that κ is indeed a context operator, i.e. κ is monotone and the resulting formula is a context formula for $\mathcal{H}$.

3.4 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. The problem is that the context formulae are passed to the decision procedure as part of the queries, so a simple syntactic caching of formulas is ineffective. However, the context consists of all discovered abstract states at the current iteration. Therefore it 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 in the analysis of 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.

3.5 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 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 [10]. In particular, the propagation is still applicable in the presence of heap manipulations that preserve the invariants in each loop-free code fragment.

4 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) simple safety properties, such as absence of null pointer dereferences and array bounds checks; (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 4.1 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 [21].

We also examined the impact of our quantifier instantiation and caching on the running time of the analysis. We have found that disabling caching slows down the analysis by 1.3 to 1.5 times, while disabling instantiation slows down the analysis by 1.2 to 3.6 times.

benchmark	used DP	# predicates	# DP calls	running time
		total (user provided)	total (cache hits)	total (DP)
List.reverse	MONA	7 (2)	369 (19%)	5s (71%)
DLL.addLast	MONA	7 (1)	156 (13%)	3s (65%)
Skiplist.add	MONA	16 (3)	770 (20%)	35s (74%)
Tree.add	MONA	11 (3)	983 (27%)	81s (91%)
ParentTree.add	MONA	11 (3)	979 (27%)	83s (89%)
SortedList.add	MONA, CVC lite	11 (3)	541 (17%)	18s (66%)
Linear.arrayInv	CVC lite	7 (5)	882 (52%)	57s (97%)

Figure 4.1: Results of Experiments

Note that our implementation of the algorithm is not highly tuned in terms of aspects orthogonal to Bohne's algorithm, such as type inference of 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. The Bohne algorithm could also infer loop invariants in such examples given the appropriate abstraction predicates.

5 Conclusions

We have presented Bohne, a data structure analysis algorithm based on symbolic shape analysis that generalizes predicate abstraction and infers Boolean algebra expressions over sets given by predicates on objects. We have shown that this abstraction can be fruitfully combined with a collection of decision procedures that operate on independent subgoals of the same proof obligation. The effect of such an approach is that the analysis synthesizes facts that are used to communicate information between different decision procedures. As a result, we were able to combine precise reasoning about reachability in tree-like structures with reasoning about first-order properties in general graphs and integer arithmetic properties. As an example that illustrates this combination, we have verified a sorted linked data structure without specializing the analysis to sorting or reachability properties.

In addition, we have deployed a range of techniques that significantly improve the running time of the analysis and the level of automation compared to direct application of the algorithm. These techniques include context-dependent finite-state abstraction, semantic caching of formulas, propagation of conjuncts, and domain-specific quantifier instantiation. Our current 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 and that its running time makes it usable for verification of such complex properties.

Bibliography

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

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

[3] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

[4] 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.

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

[6] 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.

[7] 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.

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

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

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

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

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

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

[14] 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.

[15] 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.

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

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

[18] N. D. Jones and S. S. Muchnik. *Program Flow Analysis: Theory and Applications*, chapter Chapter 4: Flow Analysis and Optimization of LISP-like Structures. Prentice Hall, 1981.

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

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

[22] V. Kuncak, S. Lahiri, R. Rugina, E. Yahav, and T. Wies. A proposal to establish shape analysis benchmarks. POPL 2006, Charleston, South Carolina, January 2006.

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

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

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

[26] 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.

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

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

[29] 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.

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

[31] 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 International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*, volume 3148 of *Lecture Notes in Computer Science*, pages 181–198. Springer, Jan. 2005. Available at http://www.cs.tau.ac.il/∼rumster/vmcai05.pdf.

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

[33] G. Nelson. Verifying reachability invariants of linked structures. In *POPL*, 1983.

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

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

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

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

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

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

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

[41] 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.

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

[43] 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.

[44] 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.

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

[46] G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *TOCL*, 2005. (to appear).

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

[48] 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.