Integrating Shape Analysis into the Model Checker BLAST

Grégory Théoduloz
gregory.theoduloz@epfl.ch

Advisors: Prof. Thomas A. Henzinger
Dr. Dirk Beyer

Master Thesis

Models and Theory of Computation
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne

Abstract

Many software model checkers are based on predicate abstraction. Values of variables in branching conditions are represented abstractly using predicates. The strength of this approach is its path-sensitive nature. However, if the control flow depends heavily on the values of memory cells on the heap, the approach does not work well, because it is difficult to find 'good' predicate abstractions to represent the heap. In contrast, shape analysis can lead to a very compact representation of data structures stored on the heap. In this thesis, we combine shape analysis with predicate abstraction, and integrate it into the software model checker BLAST. Because shape analysis is expensive, we do not apply it globally. Instead, we ensure that shapes are computed and stored locally, only where necessary for proving the verification goal. To achieve this, we extend lazy abstraction refinement, which so far has been used only for predicate abstractions, to shapes. This approach does not only increase the precision of model checking and shape analysis taken individually, but also increases the efficiency of shape analysis (we do not compute shapes where not necessary). We implemented the technique by extending BLAST with calls to TVLA, and evaluated it on several C programs manipulating data structures, with the result that the combined tool can now automatically verify programs that are not verifiable using either shape analysis or predicate abstraction on its own.

Contents

Chapter 1

Introduction

Counterexample-guided abstraction refinement [2] has dramatically increased the performance of software model checkers in the past few years, and has made it possible to verify programs that were previously too large for model checking [1]. However, current implementations of model checkers are not capable of dealing efficiently with the contents of the heap.

Shape analysis [10] is a static data-flow analysis that models the heap contents in a compressed way. It provides a finite abstraction of the portion of the program state space that is located on the heap. However, the method often produces a large amount of false positives due to its path-insensitive nature. Besides this, shape analysis is among the most expensive static analyses.

The contribution of this paper is to show how to increase the effectiveness of model checking and the efficiency of shape analysis by combining the advantages of both techniques. By computing both predicate and shape information, we increase the precision of the analysis, and thus obtain fewer false positives than either method on its own. The efficiency of pure shape analysis is improved, because expensive shape computations (such as abstract postconditions) are performed only at those control locations where the shape information is necessary to prove the verification goal. To achieve this, we apply the 'lazy abstraction' paradigm [6] to shapes. Lazy abstraction involves both lazy (on-the-fly) abstraction construction and lazy (only-where-necessary) abstraction refinement.

Lazy abstraction construction means that an abstract reachability tree (ART) for the program is computed on-the-fly. Each node of the ART is labeled with both predicate and shape information. The computation of a branch in the ART is terminated when the concrete states represented by the leaf are covered by another node in the tree. Thus, the ART construction is path-sensitive and avoids the computation of joins.

Lazy abstraction refinement means that predicate and shape information is refined only along branches of the ART that represent spurious counterexamples, in order to remove these false positives. In Blast [5], additional predicates are discovered using Craig interpolation [8]. This method allows the pin-pointing of necessary predicates to individual program locations. A key novelty of this paper is that we use interpolation-based predicate discovery also to refine the granularity of the shape analysis. Based on a computation of locally necessary predicates, in combination with an alias analysis and type information for the pointer variables, our algorithm decides, individually for each location along a spurious counterexample, which predicates and pointers to observe, and how to refine the local shape abstraction, so that the infeasible error path is removed.

We implemented this algorithm in Blast, using calls to Tvla for shape operations. We evaluated the method by applying it to several C programs that manipulate list data structures. About half of the programs could not be verified previously, neither by pure predicate-based model checking (the old version of Blast) nor by pure shape analysis (Tvla): either method on its own is not sufficiently precise and leads to false positives, while the integrated approach succeeds in automatically proving the programs correct. The other half of the programs can be verified with one of the two individual methods, but we use them to measure the overhead of our combined implementation. We found that interpolation and iterated refinement adds about 20 % to the cost of shape operations (but fewer of those are required, due to lazy analysis).

Related Work

Fischer et al. implemented in Blast a combination of a lattice-based data-flow analysis with predicate abstraction [3], but they did not consider automatic refinement of their data-flow analysis. Gulavani and Rajamani proposed a non-lazy CEGAR method for abstract interpretation, and they showed how it can be applied to shape analysis [4]. However, their refinement is done globally, not lazily, which we believe is crucial for the scalability of expensive analyses such as shape analysis. Rinetzky, Sagiv, and Yahav experimented with a method speeding up shape analysis which is based on ignoring parts of the heap by constructing procedure summaries [9]. To the best of our knowledge, the integration of shape analysis into a lazy abstraction framework is a novel contribution of this paper.

Chapter 2

Existing Techniques

2.1 Model Checking by Predicate Abstraction

2.1.1 Counterexample-guided Abstraction Refinement

The classical counterexample-guided abstraction refinement (CEGAR) algorithm starts with an initial (trivial) predicate abstraction, and refines the abstraction in every iteration. During one iteration, it explores the abstract reachability tree. If all abstract states are visited and all states are safe, the algorithm stops with answer 'safe' (and returns the abstract reachability tree as proof). If an (abstract) counterexample is found it has to be checked if there exists a feasible (concrete) path through the program (which is reported as a bug), or if the counterexample is 'spurious' due to the too coarse abstraction, i.e., there is no corresponding feasible concrete path through the program. Then the concrete path is analyzed to discover new predicates that need to be added to the abstract representation of the program, in order to eliminate the spurious counterexample in the next iteration. This is repeated until either the program is proven safe, or a program bug is found [2, 1].

2.1.2 Lazy Abstraction Refinement

The classical version of the abstract-check-refine loop has two drawbacks: first, it is not necessary to represent and analyze the state space that is not reachable, and second, it is not necessary to refine portions of the program that are already proved save. Lazy abstraction refinement integrates the steps of the abstract-check-refine loop into an on-the-fly analysis that refines the predicate abstraction locally. The algorithm produces the refinement of the predicate abstraction on demand, i.e., it discovers predicates only for a particular error path, and refines the abstraction only at the locations along the error path that need the new predicates to eliminate the error path [6].

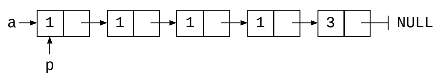
2.1.3 Craig Interpolation

The crucial measure for the efficiency of the analysis is the number of predicates in the abstraction. To keep the number of predicates per location as small as possible, interpolation-based predicate discovery can be used to produce precisely the predicates that are needed to eliminate the infeasible path in the abstract reachability tree (no more and no less). Given an error path and the corresponding path formula that was used to prove the infeasibility of the path, we wish to discover the predicates needed for one location. The path formula is split at the location into two formulas, a prefix that leads the program from the initial program location to the considered location, and the postfix that leads the program from the considered location to the error location. The *Craig interpolant* is a formula such that (1) it is implied by the prefix formula, (2) its conjunction with the postfix formula is unsatisfiable, and (3) it contains only variables that occur in the prefix formula and in the postfix formula [5, 8].

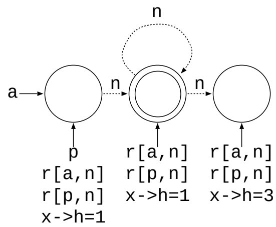
2.2 Data-Flow Analysis by Shape Analysis

Shape analysis is a static analysis that represents unbounded instances of (recursive) data structures by finite structures, called shape graphs. A shape graph is an abstraction of an instance of a heap data structure, obtained by blurring some information (e.g., about the data elements) and keeping track of the *shape* of the data structure, depending on the abstraction level of the analysis. We consider the formalism of Sagiv, et al. [7, 10], which provides a parametric framework for precise shape analysis. In this formalism, shape graphs are represented as three-valued logical structures.

Figure 2.1(a) shows an instance of a list data structure consisting of five list elements, four with data value 1 and one with data value 3. The pointers a and p point to the first list element. Figure 2.1(b) shows a shape graph that represents list instances where pointers a and p point to the first list element and all data values except the last one have data value 1, resp. 3. The list instance in Figure 2.1(a) is an instance of this shape graph. The shape graph is represented by the unary predicates $a, p, r_{p,n}, sm$, and the binary predicate n. The predicate $a(v)$ is true if the pointer variable a is pointing to node v (same for $p(v)$); the predicate $n(v, u)$ is true if the next pointer of node v is pointing to node u; the predicate $r_{p,n}(v)$ is true if node v is reachable from pointer p via the next pointer relation, and the predicate $sm(v)$ is false for a node that represents a single list element and



(a) Concrete list on heap



(b) Shape of the concrete list

Figure 2.1: Example program and two list representations

has value 1/2 for summary nodes. A summary node represents one or more list elements (drawn as double-circled nodes in the picture). E.g., the next pointer of a list element that is abstracted by the second node may point to the same or may point to the third node, and the next pointer of the first list element may not point to all list elements that are represented by the second node. The dotted edges represent the 'don't know' value (1/2) of the predicate n.

The abstract post operator is implemented as a predicate transformer. The precision of the analysis can be dramatically increased by the use of two techniques. First, *instrumentation predicates* can be added to the shape abstraction. An instrumentation predicate represents a property that can be derived from *core predicates* (e.g., reachability). They have an associated formula which is expressed in first order logic (with transitive closure) over core predicates. In abstract shape graphs, those predicates may have a more precise value than the one computed from the abstract values of the core predicates, resulting in increased precision. Second, instead of using only predicate update formulas when computing an abstract post, the two complementary operations *focus* and *coerce* are applied before (resp. after) applying the update formulas. Given a set of *focus formulas*, the operation *focus* yields an equivalent set of shape graphs, in which the focus formulas have definite values (0 or 1). In particular, this operation enables materialization of summary nodes. Given constraints derived from instrumentation predicates and from concrete semantics, the operation *coerce* rules out infeasible shape graphs, and replaces some indefinite values (1/2) by precise ones (0 or 1).

Chapter 3

Overview and Example

3.1 CEGAR with Shapes

The classical CEGAR algorithm is extended by a heap abstraction, i.e., the abstraction consists of a predicate abstraction *and* a heap abstraction (cf. Figure 3.1). The initial predicate abstraction is the trivial predicate abstraction (only predicate *true*), and the initial heap abstraction is the trivial shape class (representing every heap).

If the complete abstract reachability tree is explored and no abstract state is unsafe, the algorithm stops and answers 'safe'. If an error path is found, the path formula as described in [5] is constructed and checked for satisfiability. If the path formula is unsatisfiable, then the infeasibility is due to the *predicate abstraction*, and the interpolation procedure will discover new predicates that are added to the predicate abstraction to avoid this infeasible path in the next iteration. The heap abstraction is not changed.

If the path formula is satisfiable, it may be due to its incompleteness in the presence of recursive data structures. Therefore, this does not necessarily mean that a bug is found. We construct the (more precise) *extended path formula* that takes also into account the may-aliasing relation that can occur over nodes. If the generated extended path formula is feasible, then the system is considered unsafe; otherwise, we use the interpolation procedure for the new extended path formula, and use the interpolant predicates to decide on how to refine the heap abstraction.

3.2 Example

The function in Figure 3.2 generates first a list that contains a sequence of data values either 1 or 2 —depending on a given variable `flag`—, and that ends with data value 3. The second part of the function verifies that the list

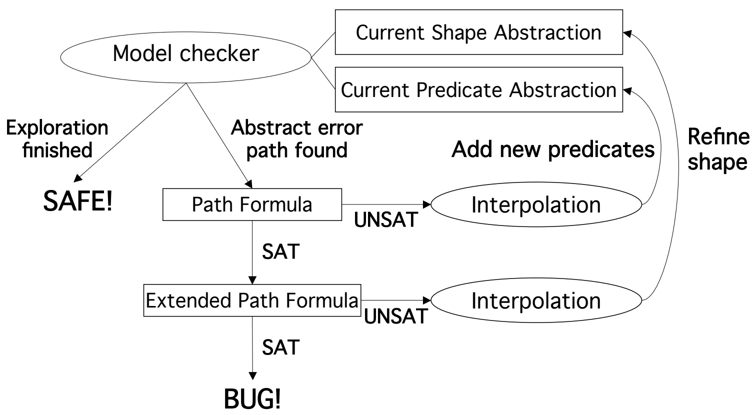


Figure 3.1: Abstraction refinement with heap abstraction

```
typedef struct node {
  int h;
  struct node *n;
} *List;
void foo(int flag) {
  List a = (List) malloc(...);
  if (a == NULL) exit(1);
  List p = a;
  while (random()) {
    if (flag)  p->h = 1;
    else       p->h = 2;
    p->n = (List) malloc(...);
    if (p->n == NULL) exit(1);
    p = p->n;
  }
  p->h = 3;
  /* Check it */
  p = a;
  if (flag)
    while (p->h == 1)  p = p->n;
  else
    while (p->h == 2)  p = p->n;
  assert(p->h == 3);
}
```

Figure 3.2: Example C program

really consists of a sequence of data values 1 or 2 —again depending on the flag—, and that it ends with data value 3.

Path-insensitive *static analysis* cannot prove this program safe, because after the if statement in the first while loop the analysis forgets the fact that the values in the list depend on the flag. This is due to the join that would occur in the corresponding shape lattice. Path-sensitive *predicate-based reachability analysis* cannot prove this program safe either, because the analysis does not keep track of the heap, i.e., which values are stored in the list. The *combination of predicate abstraction and shape analysis* tracks

Command	Constraint
$1 : a := malloc()$	$true$
$2 : pred(a \neq 0)$	$\langle a, 1\rangle \neq 0$
$3 : p := a$	$\langle p, 3\rangle = \langle a, 1\rangle \wedge \langle\langle p, 3\rangle\text{->}h, 3\rangle = \langle\langle a, 1\rangle\text{->}h, 1\rangle$
$4 : p\text{->}h := 3$	$\langle\langle p, 3\rangle\text{->}h, 4\rangle = 3$
	$\wedge (\langle p, 3\rangle = \langle a, 1\rangle \Rightarrow \langle\langle a, 1\rangle\text{->}h, 4\rangle = 3)$
$5 : p := a$	$\langle p, 5\rangle = \langle a, 1\rangle \wedge \langle\langle p, 5\rangle\text{->}h, 5\rangle = \langle\langle a, 1\rangle\text{->}h, 4\rangle$
$6 : pred(flag = 0)$	$\langle flag, 0\rangle = 0$
$7 : pred(p\text{->}h \neq 2)$	$\langle\langle p, 5\rangle\text{->}h, 5\rangle \neq 2$
$8 : pred(p\text{->}h \neq 3)$	$\langle\langle p, 5\rangle\text{->}h, 5\rangle \neq 3$
$9 :$ ERROR	

Interpolants: $\langle p, 3\rangle = \langle a, 1\rangle$, $\langle\langle a, 1\rangle\text{->}h, 4\rangle = 3$, $\langle\langle p, 5\rangle\text{->}h, 5\rangle = 3$

Figure 3.3: Extended path formula for the first infeasible error path.

both predicate and shape information at the same time. When computing the successor of an abstract region, the method computes the successor for each of the two abstractions, checks that the successor region is non-empty, and ensures that the two abstract region do not contradict each other. The analysis starts with the trivial predicate abstraction and the trivial heap abstraction.

3.2.1 First Refinement Step

The first (infeasible) error path that our new method reports skips the first while loop, sets `p->h=3`, assumes `flag=0`, skips the while loop of the else branch and violates the assertion. The list consists of one list element: $\langle 3\rangle$. The method described in [5] does not report this counterexample as spurious due to its limitation to pointers and integers, and the way in which memory cells are encoded. Consequently, the extended path formula for this example is computed. The extended path formula is given in Figure 3.3. For clarity reason, non-relevant parts of the formula have been omitted. The number annotated to a value in a path formula corresponds to the number of the command that may have written this value. Such a numbering encodes the history of computation along the path. The analysis of the error path (more precisely, the analysis of the interpolants of the (unsatisfiable) extended path formula) yields that we have to track the list pointed to by pointers `a` and `p`, and the node predicate `x->h=3` (where `x` is the parameter of the parametric predicate) for this list, i.e., we have to choose the shape type according to the C type of pointer variable `a` (or `p`) track the shape for the data structure that `a` is pointing to, and finally we add the node predicate `x->h=3` to the *shape abstraction*. In addition, alias analysis yields that no other pointers than `a` and `p` need to be tracked.

Command	Constraint
$1 : a := malloc()$	$true$
$2 : pred(a \neq 0)$	$\langle a, 1\rangle \neq 0$
$3 : p := a$	$\langle p, 3\rangle = \langle a, 1\rangle \wedge \langle\langle p, 3\rangle\text{->}h, 3\rangle = \langle\langle a, 1\rangle\text{->}h, 1\rangle$
	$\wedge \langle\langle p, 3\rangle\text{->}n, 3\rangle = \langle\langle a, 1\rangle\text{->}n, 1\rangle$
$4 : pred(flag = 0)$	$\langle flag, 0\rangle = 0$
$5 : p\text{->}h := 2$	$\langle\langle p, 3\rangle\text{->}h, 5\rangle = 2 \wedge \langle\langle a, 1\rangle\text{->}h, 5\rangle = 2$
$6 : p\text{->}n := malloc()$	$-$
$7 : pred(p\text{->}n \neq 0)$	$-$
$8 : p := p\text{->}n$	$-$
$9 : p\text{->}h := 3$	$-$
$10 : p := a$	$\langle p, 10\rangle = \langle a, 1\rangle \wedge \langle\langle p, 10\rangle\text{->}h, 10\rangle = \langle\langle a, 1\rangle\text{->}h, 5\rangle$
	$\wedge \langle\langle p, 10\rangle\text{->}n, 10\rangle = \langle\langle a, 1\rangle\text{->}n, 1\rangle$
$11 : pred(flag = 0)$	$\langle flag, 0\rangle = 0$
$12 : pred(p\text{->}h \neq 2)$	$\langle\langle p, 10\rangle\text{->}h, 10\rangle \neq 2$
$13 : pred(p\text{->}h \neq 3)$	$\langle\langle p, 10\rangle\text{->}h, 10\rangle \neq 3$
14 : ERROR	

Interpolants: $\langle p, 10\rangle = \langle a, 1\rangle$, $\langle\langle a, 1\rangle\text{->}h, 5\rangle = 2$, $\langle\langle p, 10\rangle\text{->}h, 10\rangle = 2$

Figure 3.4: Extended path formula for the second infeasible error path

(a) In second error path (b) In third error path

Figure 3.5: Shape graphs when ERROR is reached

3.2.2 Second Refinement Step

The second (infeasible) error path enters the first while loop, assumes `flag=0`, sets `p->h=2`, sets `p->h=3`, assumes `flag=0`, skips the while loop of the else branch and violates the assertion. The list represents the sequence $\langle 2, 3\rangle$. The abstract state region associated with the program location before the assertion is represented by the predicate *true* on the one hand, and the shape graph in Figure 3.5(a) on the other hand. The current shape class knows the node predicate `x->h=3`, but not the node predicate `x->h=2`, and therefore consists of two nodes, the first representing a list element with data value $\neq 3$ (node predicate `x->h=3` is *false*) and the second a list element with data value 3. The extended path formula for this error path is given in Figure 3.4 (some aliasing constraints are omitted for clear presentation). Since the path formula is unsatisfiable, we know that the path is infeasible. To proceed, we add the node predicate `x->h=2` to the *shape abstraction* based on the interpolants.

3.2.3 Third Refinement Step

The third (infeasible) error path enters the first while loop, assumes `flag=1`, sets `p->h=1`, sets `p->h=3`, assumes `flag=0`, skips the while loop of the else branch and violates the assertion. The list represents the sequence $\langle 1, 3\rangle$. The abstract state region associated with the program location before the assertion is represented by the predicate *true* on the one hand, and the shape graph in Figure 3.5(b) on the other hand. The current shape graph knows the node predicates `x->h=3` and `x->h=2`, and therefore consists of two nodes, the first representing a list element with data value 2 (node predicate `x->h=2` is *true*) and the second a list element with data value 3. But the predicate abstraction does not keep track of predicate `flag`, which leads to the infeasible situation that in the first while loop the predicate is assumed to be *true* and in the second part of the program the same predicate is assumed to be *false*. To proceed, we add the boolean predicate `flag` to the *predicate abstraction*.

3.2.4 Fourth Refinement Step

The fourth (infeasible) error path enters the first while loop, assumes `flag=1`, sets `p->h=1`, sets `p->h=3`, assumes `flag=1`, skips the while loop of the *then* branch and violates the assertion. The list represents the sequence $\langle 1, 3\rangle$. We add the node predicate `x->h=1` to the *shape abstraction*.

3.2.5 Final Iteration

The last iteration re-visit the nodes that need to be refined and unfolds the remaining states or marks them covered, and thus constructs the complete reachability tree that acts as *safety certificate* since the error node does not belong to it. Figure 3.6 gives the shape graphs associated to the several occurrences of the location at line 17 (after `p->h = 3;`) in the complete reachability tree when `flag` is assumed to be *true*. It is interesting to note that taking the join of all these shape graphs result in the fix-point that a predicated data-flow analysis would have computed for that location.

If the program contained a second list that is created but never checked, then the analysis would not track the shapes of that list, because the interpolants yield only predicates that are inevitable for eliminating the infeasible error path.

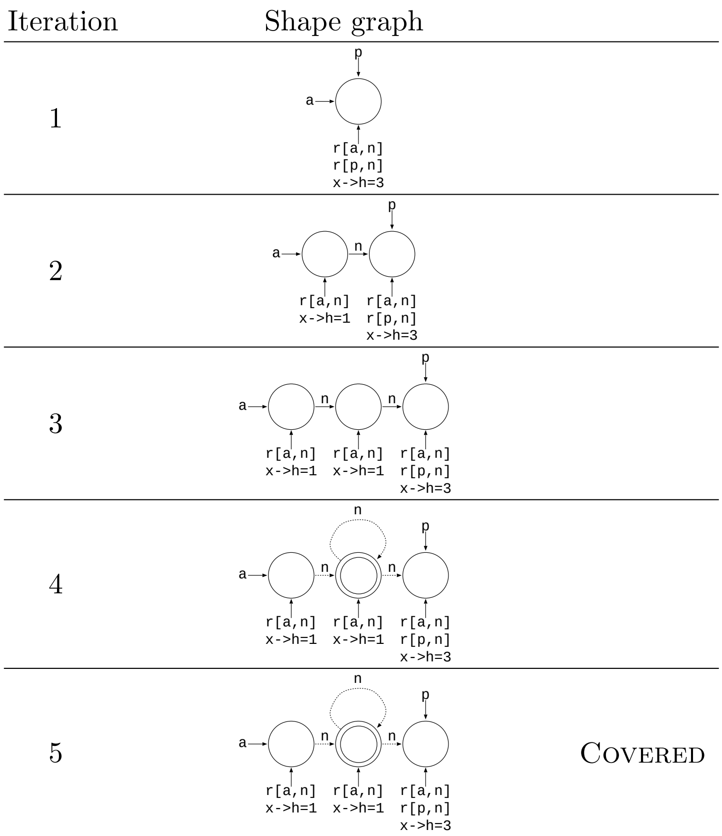


Figure 3.6: Shape graphs at line 17 occurring in the reachability tree for nodes where `flag` is assumed to be *true*

Chapter 4

Lazy Abstraction Refinement of Shapes

4.1 Shape Abstraction

4.1.1 Shape Classes

The level of abstraction of the shape analysis is defined by a *shape class* $\mathbb{S} = (\mathcal{P}_{core}, \mathcal{P}_{instr}, \mathcal{P}_{abs})$, which consists of three sets of predicates: (1) a set $\mathcal{P}_{core}$ of core predicates, (2) a set $\mathcal{P}_{instr}$ of instrumentation predicates with $\mathcal{P}_{core} \cap \mathcal{P}_{instr} = \varnothing$, where every instrumentation predicate $p \in \mathcal{P}_{instr}$ has an associated defining formula φ^p over core predicates, and (3) a set $\mathcal{P}_{abs} \subseteq \mathcal{P}_{core} \cup \mathcal{P}_{instr}$ of abstraction predicates. The set of all predicates of the shape class is denoted by $\mathcal{P} = \mathcal{P}_{core} \cup \mathcal{P}_{instr}$.

The set of core predicates must contain the special unary predicate sm which has the value 0 for normal nodes and 1/2 for summary nodes. Moreover, we distinct two special subsets of the core predicates: the set $\mathcal{P}_{pt}$ of points-to predicates and the set $\mathcal{P}_{node}$ of node predicates. A *points-to predicate* $pt_{\mathtt{x}}(v)$ is a unary predicate that indicates whether the pointer variable `x` points to node v. A *node predicate* $npred_p(v)$ is a unary predicate that corresponds to some boolean predicate p (from the predicate abstraction) that holds for a variable that points to node v. The boolean predicate p is parametric on some variable name. We denote by $p[\mathtt{x}]$ an instance of the predicate p that refers to variable `x`. Node predicates represent the content of a structure element, rather than the structure of the shape itself.

A shape class $\mathbb{S}$ *refines* a shape class $\mathbb{S}'$, written $\mathbb{S} \preccurlyeq \mathbb{S}'$, if (1) $\mathcal{P}'_{core} \subseteq \mathcal{P}_{core}$, (2) $\mathcal{P}'_{instr} \subseteq \mathcal{P}_{instr}$, and (3) $\mathcal{P}'_{abs} \subseteq \mathcal{P}_{abs}$. The *union* of two shape classes $\mathbb{S}$ and $\mathbb{S}'$ is the shape class $(\mathcal{P}_{core} \cup \mathcal{P}'_{core}, \mathcal{P}_{instr} \cup \mathcal{P}'_{instr}, \mathcal{P}_{abs} \cup \mathcal{P}'_{abs})$ (w.l.o.g., we require $\mathcal{P}_{core} \cap \mathcal{P}'_{instr} = \varnothing$ and $\mathcal{P}_{instr} \cap \mathcal{P}'_{core} = \varnothing$).

A *shape graph* $s = (V, \mathit{Val})$ of a shape class $\mathbb{S} = (\mathcal{P}_{core}, \mathcal{P}_{instr}, \mathcal{P}_{abs})$ consists of a set of shape nodes V and a valuation of the predicates (in a three-valued logic) over V: for a predicate p in $\mathcal{P}_{core} \cup \mathcal{P}_{instr}$ of arity n, $\mathit{Val}(p) : V^n \to \{0, 1, 1/2\}$.

4.1.2 Shape Regions

A *shape region* consists of a shape class $\mathbb{S}$ and a set S of shape graphs. Given a shape class $\mathbb{S}$, the shape region $\top_{\mathbb{S}} = (\mathbb{S}, \{s_{1/2}\})$ includes all possible shape regions (corresponding to *true* in the predicate abstraction), where $s_{1/2}$ is the shape graph with a single shape node and the constant function $1/2$ as valuation for every predicate. The shape region $\bot_{\mathbb{S}} = (\mathbb{S}, \varnothing)$ corresponds to *false* in the predicate abstraction.

Let $\mathbb{S}$ and $\mathbb{S}'$ be two shape classes such that $\mathbb{S} \preccurlyeq \mathbb{S}'$. A shape graph s' of shape class $\mathbb{S}'$ can be extended to the shape graph $s = \tau_{\mathbb{S}' \triangleright \mathbb{S}}(s')$ of shape class $\mathbb{S}$ such that the set of shape nodes is left unchanged ($V = V'$), and for each predicate p in $\mathcal{P} \setminus \mathcal{P}'$, the value of p is $1/2$ for all shape nodes. We extend the operator τ to sets of shape graphs in the natural way. A shape region $(\mathbb{S}, S)$ is *covered* by a shape region $(\mathbb{S}', S')$, denoted by $(\mathbb{S}, S) \sqsubseteq (\mathbb{S}', S')$, if $\tau_{\mathbb{S}' \triangleright (\mathbb{S} \cup \mathbb{S}')}(S') = \tau_{\mathbb{S} \triangleright (\mathbb{S} \cup \mathbb{S}')}(S) \sqcup \tau_{\mathbb{S}' \triangleright (\mathbb{S} \cup \mathbb{S}')}(S')$, where $\sqcup$ is the join of two sets of shape graphs as defined in Tvla [7].

The *abstract semantics* $\mathsf{SP}_{\mathbb{S}}$ is defined by $\mathsf{SP}_{\mathbb{S}}((\mathbb{S}, S), \mathsf{op}) = (\mathbb{S}, [\![\mathsf{op}]\!](S))$, where $[\![.]\!]$ is defined as in Tvla [7]. Depending on the operations, we apply Tvla's operators *focus* and *coerce* before (respectively after) transforming a set of shape graphs.

4.2 Extracting Interpolants from Extended Path Formulas

For a more precise analysis of the memory configuration, we extend the path formulas that were previously used in Blast to recursive data structures.

4.2.1 Programs, Lvalues, Paths and Path Formulas

Our formalization of programs is similar to [5]. A program is represented as a set of control flow automata, a path t of length tsize is a sequence $\mathsf{op}_1; \ldots; \mathsf{op}_{\mathit{tsize}}$ of commands, which can be either statements or assume predicates. In the rest of this paper, we consider flat programs (i.e., program with a single function). Our approach can be extended to programs with several functions. The program variables are either integer values or pointers to (possibly recursive) structures with fields that are integers and

$$
\begin{array}{lcl}
\textit{lvalue} & ::= & \textit{ident} \mid \textit{ident->field} \\
\textit{command} & ::= & \textit{statement} \mid \textit{predicate} \\
\textit{statement} & ::= & \textit{ident} := \textit{expression} \\
& \mid & \textit{ident} := \textit{alloc}() \\
& \mid & \textit{ident} := \textit{ident} \\
& \mid & \textit{ident} := \textit{ident->field} \\
& \mid & \textit{ident->field} := \textit{ident} \\
\textit{predicate} & ::= & \text{FOL formula over } \textit{ident}\text{s (variables)}
\end{array}
$$

Figure 4.1: Grammar of a program

pointer to structures. We restrict lvalues that can occur in a program to *ident* and *ident->field*, where *ident* denotes a variable identifier and *field* denotes a name of a structure field. The function F maps an lvalue to the set of labels of the structure pointed by the lvalue if the lvalue has a pointer type, and to an empty set if the lvalue has an integer type. The statements and predicates composing a program are given in Figure 4.1.

The semantics for a path is given in terms of the strongest postcondition operator: if the formula φ represents a state of the program and op is a command, then the formula $\mathsf{SP}.\varphi.\mathsf{op}$ represents the set of successor states. The predicate abstraction for a path is given by a mapping $\Pi : [1..\textit{tsize}] \rightarrow 2^{FOL}$ from path locations to sets of atomic predicates. For a formula φ, the abstraction w.r.t. a set of atomic predicates P is the strongest formula φ' with atomic predicates from P such that φ implies φ'. The operator SP_Π is the abstraction of the operator SP, i.e., the formula $\mathsf{SP}_\Pi.\varphi.\mathsf{op}_i$ is the abstraction w.r.t. $\Pi(i)$ of the formula $\mathsf{SP}.\varphi.\mathsf{op}_i$. We extend SP and SP_Π to paths in the natural way. A path t is *SP-infeasible* (*SP_Π-infeasible*) if $\mathsf{SP}.\textit{true}.t$ ($\mathsf{SP}_\Pi.\textit{true}.t$) is not satisfiable.

To check whether a given error path is feasible (i.e., there exists a corresponding feasible execution of the program), we construct a *path formula* (PF), which is the conjunction of several constraints, one per instruction, such that the PF is feasible iff the path is feasible. The technique for building PFs from [5] cannot be reused directly, because it is restricted to programs without recursive data structures. Also, that approach cannot be extended trivially because it would result in infinite formulas. However, since the number of memory cells possibly involved in the path formula is bounded, we can produce a finite, sound and complete path formula. The address of each structure on the heap that is accessed on the path, was previously assigned to a pointer variable at some point, because we consider a restricted set of possible lvalues. To be able to refer to those addresses in our constraint formulas, we use SSA-like renamed lvalues.

4.2.2 Lvalue Constants, Annotated Lvalues and Aliasing

An *lvalue constant* is either $\langle ident, l\rangle$ (*variable constant*) or $\langle\langle ident, l\rangle$->$field, l'\rangle$ with $l, l' \in [0..tsize]$ and $l' \geq l$. An *annotated lvalue* is either *ident* or $\langle ident, l\rangle$->*field*. The *labels* l and l' correspond to the position in the path where the annotated values *may* have been modified. The function Clean maps an lvalue constant or an annotated lvalue to the lvalue by removing the labels. An *annotated lvalue map* θ is a function from annotated lvalues to numbers. The *lvalue renaming function* Sub.$\theta.v$ is defined by $\mathsf{Sub}.\theta.p = \langle p, \theta(p)\rangle$ and $\mathsf{Sub}.\theta.(p\text{->}f) = \langle(\mathsf{Sub}.\theta.p)\text{->}f, \theta((\mathsf{Sub}.\theta.p)\text{->}f)\rangle$ (p is a variable and f is a field).

To encode into the path formula the aliasing among memory cells, we use the function may that maps a position in the path and an lvalue constant to the set of variable constants that may have the same value (i.e., $\langle p, l_p\rangle \in \mathsf{may}.l.c$ if, after the l-th command of the path, the value of c may be equal to the value of p_1 after the l_1-th command on the path).

4.2.3 Path Formulas and Constraints

The function Con maps a pair (θ, Γ) consisting of an annotated lvalue map θ and a constraint map $\Gamma : \mathbb{N} \to FOL$, and a command op_i, to a pair (θ', Γ') consisting of a new annotated lvalue map and a new constraint map. Given a path, we compute recursively the result of Con along the path by computing $(\theta_l, \Gamma_l) = \mathsf{Con}.(\theta_{(l\text{-}1)}, \Gamma_{(l\text{-}1)}).\mathsf{op}_l$ (where l is the location of op_l in the path). The map θ_0 is a constant map to 0 and Γ_0 is the empty map. The map θ_l differs from $\theta_{(l\text{-}1)}$ only for annotated lvalues that may be modified by op_l, which are mapped to l by θ_l. The map Γ_l results from the map $\Gamma_{(l\text{-}1)}$ extended by mapping l to the constraint derived from op_l. We derive the constraints from path commands similarly to [5]. A major extension is necessary for assignments to pointers. Since the structure may be recursive, we cannot 'unroll' the data structure to equate all possibly reachable memory cells, because this yields infinite formulas. Additionally, we have to add aliasing constraints for cases where several lvalue constants may point to the same memory cell. The formal definition of the function Con is given in Figure 4.2. The path formula is obtained by taking the conjunction of all formulas in the final constraint map. Note that the size of the formula is highly dependent in the precision of the alias analysis.

The definition of Con refers to the following two functions. Given two variables and two associated annotated lvalue maps, the function eqvar returns

Algorithm 1 *Extract*(t)

Input: an infeasible path $t = (\mathsf{op}_1 : pc_1); \ldots; (\mathsf{op}_n : pc_n)$
Output: a map Π from the locations of t to sets of atomic predicates
$\Pi.pc_i := \varnothing$ for $1 \leq i \leq n$
$(\cdot, \Gamma) := \mathsf{Con}.(\theta_0, \Gamma_0).t$
$\mathbb{P} :=$ derivation of $\bigwedge_{1 \leq i \leq n} \Gamma.i \vdash \mathit{false}$
for $i := 1$ to n **do**
  $\varphi^- := \bigwedge_{1 \leq j \leq i} \Gamma.j$
  $\varphi^+ := \bigwedge_{i+1 \leq j \leq n} \Gamma.j$
  $\psi := \textsc{Itp}(\varphi^-, \varphi^+)(\mathbb{P})$
  $\Pi.pc_i := \Pi.pc_i \cup \mathsf{Atoms}(\mathsf{Clean}(\psi))$
return Π

a constraint corresponding to the equality of two variables considering their fields (if any).

$$\begin{aligned}\mathsf{eqvar}.(s_1, \theta_1).(s_2, \theta_2) = {} & (\mathsf{Sub}.\theta_1.s_1 = \mathsf{Sub}.\theta_2.s_2) \\ & \wedge \bigwedge_{f \in \mathsf{F}(s_1)} (\mathsf{Sub}.\theta_1.(s_1\text{->}f) = \mathsf{Sub}.\theta_2.(s_2\text{->}f))\end{aligned}$$

Given a boolean and a FOL predicate over variables, the function clos^* returns the constraint corresponding to a predicate.

$$\mathsf{clos^*}.\theta.b.p = \begin{cases} (\mathsf{clos^*}.\theta.b.p_1)\ \mathsf{op}\ (\mathsf{clos^*}.\theta.b.p_1) & \text{if } p \equiv (p_1\ \mathsf{op}\ p_2) \\ \neg(\mathsf{clos^*}.\theta.\neg b.p_1) & \text{if } p \equiv (\neg p_1) \\ \mathsf{eqvar}.(v_1, \theta).(v_2, \theta) & \text{if } p \equiv (v_1 = v_2) \\ & \quad \text{and } b \equiv \mathit{true} \\ \mathsf{Sub}.\theta.p & \text{otherwise} \end{cases}$$

Algorithm. Algorithm 1 first constructs the constraint map (using function Con) that represents the path formula for the given path t. Then it splits the (infeasible) path formula at every program location and computes the predicates that are necessary to eliminate the infeasible error path, for refining the abstraction in a way that makes the abstract path also infeasible. For a given split of the path formula into φ^- and φ^+, and a proof $\mathbb{P}$ of unsatisfiability of $\varphi^- \wedge \varphi^+$, the function $\textsc{Itp}(\varphi^-, \varphi^+)(\mathbb{P})$ returns the interpolant formula ψ for the proof $\mathbb{P}$ and the formulas φ^- and φ^+. The function Atoms returns the set of atomic predicates of a formula.

Theorem 1 (Soundness). Let t be a path of a program P. The path t is SP-infeasible iff t is SP_Π-infeasible for $\Pi = \mathit{Extract}(t)$.

The difference to the corresponding theorem in [5] is that our new theorem does not require the program to be free of recursive data structures. In particular, the theorem states that our method is *sound*, i.e., our method does not report infeasibility although a real bug exists. However, the theorem does not state that our method is necessarily *complete*. There are cases where we cannot eliminate an infeasible path by refinement of the abstraction or of the shape class. This is a general limitation of shape analysis with a fixed set of shape classes as implemented in TVLA [7], not of our refinement method.

4.3 Interpolant-based Shape Class Refinement

For a given program, we restrict the analysis to a finite set of shape classes that can be used to analyze such a program. We define thereafter the space of shape classes that our approach considers and the way in which refinement among shape classes occur.

4.3.1 Tracking Definition and Shape Types.

A *tracking definition* represents the pointers and predicates about the heap that we track while analyzing the program. A *tracking definition* consists of the following three sets: (1) the set T of *tracked pointers*, which is the set of pointer variables that may be pointing to some node in the shape, (2) the set $T_s \subseteq T$ of *separating pointers*, which is the set of variables for which we want the corresponding points-to predicates to be an abstraction predicate, and (3) the set P of *node predicates*. We define a refinement relation for tracking definitions. A tracking definition (T, T_s, P) *refines* a tracking definition (T', T'_s, P') if $T' \subseteq T$, $T'_d \subseteq T_s$, and $P' \subseteq P$.

A *shape type* $\mathbb{T}$ consists of a C structure type and a map from tracking definitions to shape classes, where the map preserves the refinement relation. For instance, a shape type for singly-linked lists could be associated with the C type `struct node {int data; struct node *next;};`, and it would map a given tracking definition (T, T_s, P) to the shape class with the following predicates: the default unary predicate *sm*, a binary predicate *next* for representing links between nodes in the list, for each variable in T a points-to predicate, which is an abstraction predicate only for variables in T_s, and the node predicates from P. More precise shape types for singly-linked list can be defined by adding instrumentation predicates for tracking, e.g., reachability and cyclicity.

4.3.2 Refinement

In Chapter 3 we described the overall algorithm (cf. Figure 3.1) of our combined approach. The remaining step we need to explain is how to refine the shape abstraction during the abstract reachability algorithm. As predicate abstraction starts with the empty set of predicates, lazy shape analysis starts with the empty tracking definition.

Consider the shape type $\mathbb{T}$. The current tracking definition is refined, if the *extended path formula is* unsatisfiable, and a variable p that occurs in an interpolant matches the C type of shape type $\mathbb{T}$. For all such variables p, we refine the current tracking definition as follows:

- We add p to the set of tracked pointers and to the set of separating pointers. We close the set of tracked pointers under aliasing.
- We add the atomic boolean predicates from the interpolants in which a tracked pointer is dereferenced, to the node predicates.

The map of shape type $\mathbb{T}$ maps the refined tracking definition to a shape class. Since the mapping preserves the refinement relation, the new shape class is a refinement of the current shape class.

The outcome of this refinement can be either (1) the infeasible error path is eliminated in the next iteration of the abstract reachability analysis, or (2) the refinement reaches a fixed point, i.e., we already have all pointers and all node predicates extracted from the path formula, and the infeasible error path occurs still in the next iteration. In the former case, the refinement succeeds and the algorithm proceeds with the refined shape abstraction. In the latter case we conclude that the shape type is not precise enough and we choose a refined shape type, and the analysis is re-launched with the new shape type.

Since the interpolation-based analysis precisely locates where refinement is necessary, we can restrict the refinement of the shape analysis to a local context, as done in [5] for predicate abstraction refinement. Also, this technique ensures that the algorithm never refines more than necessary.

Command op_l	New map θ' and $\mathsf{allocated}'$	Constraint $\Gamma'(l)$
$s := \mathit{expr}$	$\theta'(s) = l$	$\mathsf{Sub}.\theta'.s = \mathsf{Sub}.\theta.\mathit{expr}$
$s_1 := s_2$	$\theta'(s_1) = l$ $\forall f \in \mathsf{F}(s_1) : \theta'(\langle s_1, l\rangle\text{->}f) = l$	$\mathsf{eqvar}.(s_1, \theta').(s_2, \theta)$
$s_1 := s_2\text{->}f$	$\theta'(s_1) = l$ $\forall f \in \mathsf{F}(s_1) : \theta'(\langle s_1, l\rangle\text{->}f) = l$	$\mathsf{Sub}.\theta'.s_1 = \mathsf{Sub}.\theta.(s_2\text{->}f)$ $\wedge \bigwedge_{c \in \mathsf{may}.(l\text{-}1).(\mathsf{Sub}.\theta.(s_2\text{->}f))} (\mathsf{Sub}.\theta.(s_2\text{->}f) = c) \Rightarrow \mathsf{eqvar}.(s_1, \theta').(c, \theta)$
$s_1\text{->}f := s_2$	$\theta'(\langle s_1, \theta(s_1)\rangle\text{->}f) = l$ $\forall c \in \mathsf{may}.(l\text{-}1).\langle\langle s_1, \theta(s_1)\rangle\text{->}f, l\rangle :$ $\quad \forall f \in \mathsf{F}(c) : \theta'(\langle c, l\rangle\text{->}f) = l$ $\forall c \in \mathsf{may}.(l\text{-}1).\langle s_1, \theta(s_1)\rangle :$ $\quad \theta'(c\text{->}f) = l$	$\mathsf{Sub}.\theta'.(s_1\text{->}f) = \mathsf{Sub}.\theta.s_2$ $\wedge \bigwedge_{c \in \mathsf{may}.(l\text{-}1).(\mathsf{Sub}.\theta'.(s_1\text{->}f))} \begin{pmatrix} \mathsf{ite}.(c = \mathsf{Sub}.\theta'.(s_1\text{->}f)) \\ \quad .(\mathsf{eqvar}.(c, \theta').(s_2, \theta)) \\ \quad .(\mathsf{eqvar}.(c, \theta').(c, \theta)) \end{pmatrix}$ $\wedge \bigwedge_{c \in \mathsf{may}.(l\text{-}1).\mathsf{Sub}.\theta'.s_1} \begin{pmatrix} \mathsf{ite}.(c = \mathsf{Sub}.\theta'.s_1) \\ \quad .(\mathsf{Sub}.\theta'.(c\text{->}f) = \mathsf{Sub}.\theta.s_2) \\ \quad .(\mathsf{Sub}.\theta'.(c\text{->}f) = \mathsf{Sub}.\theta.(c\text{->}f)) \end{pmatrix}$
$s := \mathit{alloc}()$	$\theta'(s) = l$ $\forall f \in \mathsf{F}(s) :\ \theta'(\langle s, l\rangle\text{->}f) = l$ $\mathsf{allocated}' = \mathsf{allocated} \cup \{\langle s, l\rangle\}$	$\bigwedge_{a \in \mathsf{allocated}} (\langle s, l\rangle \neq a)$
$\mathit{predicate}(p)$		$\mathsf{clos^*}.\theta.\mathit{true}.p$

Figure 4.2: Definition of Con for each command. $(\theta', \Gamma') = \mathsf{Con}.(\theta, \Gamma).l.\mathsf{op}_l$

Chapter 5

Evaluation

5.1 Examples

We evaluated our method on six example C programs that manipulate list data structures containing integers as data elements. The programs `simple` and `simple_backw` both create a list of an arbitrary number of 1s and traverse it to check that every element is a 1. The difference between the two is the order in which the nodes are created.

The program `list` creates a list that begins with an arbitrary number of 1s, proceeds with an arbitrary number of 2s, and ends with a 3. Then, the list is traversed to check that the numbers occur in the correct order. The program `list_flag` builds a list that begins either with 1s or 2s depending on a flag, and ends with a 3, then the lists are traversed checking that the expected numbers are found. To prove safety, this example (and the following two) requires to track simultaneously a boolean predicate ($flag = 0$) and shape graphs.

The program `alternating` is similar to `list` except that the list begins with alternating 1s and 2s, and ends with a 3. The program `splice` builds the same list as `alternating`. Then, the list is split into two different lists: the first list contains the nodes at odd positions and the second list contains nodes at even positions of the original list, without the last 3. Each new list is then checked whether it contains only the same number.

5.2 Implementation

The concepts presented in this paper are implemented in Blast version 3.0, which integrates Tvla for shape transformation and the foci library of Blast 2.0 for the predicate interpolation. Tvla (written in Java) is integrated into Blast (written in OCaml) as a particular implementation of a

Program	CFA nodes	LOC	Pred. abstr.	Shape analysis	PA & SA
`simple`	26	44	FP 0.16 s (0)	0.48 s	0.50 s (1)
`simple_backw`	19	39	FP 0.36 s (4)	0.43 s	0.60 s (5)
`list`	34	54	FP 0.15 s (0)	3.74 s	3.80 s (3)
`list_flag`	35	62	FP 0.15 s (0)	FP 0.26 s	0.78 s (4)
`alternating`	30	58	FP 0.20 s (1)	FP 0.26 s	1.15 s (5)
`splice`	42	84	FP 0.68 s (3)	FP 0.66 s	6.98 s (7)

Table 5.1: Time for verifying singly-linked list manipulation programs in seconds on a 3 GHz Intel Xeon processor (CFA = control flow automaton, LOC = lines of code, FP = false positive, the number of refinement steps is given in parenthesis)

shape analysis module, so that, in principle, we are able to plug-in other shape analysis tools. The shape analysis is plugged-in to BLAST's on-the-fly analysis by extending the abstract state region, which was a triple so far (program counter, stack, predicate), by a shape region. We previously tried to integrate the shape analysis as *predicated lattice* —as described in [3]— but this method did not work well for the refinement, because the data-flow lattices are always joined at join points in the control-flow graph if the predicate regions are not different resulting in irrelevant spurious counterexamples. We rather want to distinguish the states reached on different paths (unless covered), for a more precise (more control-flow sensitive) analysis.

Table 5.1 reports the results of our experiments. None of the programs was successfully verified by BLAST's predicate abstraction without shape analysis: the system is not able to prove the program safe; rather it reports a false positive (column four in the table). Three examples can be proved safe by pure shape analysis (without predicate refinement and with tracking maximal shape information everywhere, like in TVLA), but for the other three it fails due to missing control-flow sensitivity (column five).

The model checker BLAST with *lazy shape analysis* proves all example programs safe (last column). The run-times show that the overhead for the refinement of the shape abstraction for the first three programs (compared to pure shape analysis) does not significantly increase the run-time of the analysis in these cases. In contrast, for the other three programs for which the combination of shape analysis and predicate refinement is really necessary, the reported run-time is much higher, because the other analyses are fast in finding a false positive. Not surprisingly, the run-times for `list` and `splice` are higher than the others, because their shape analysis is more involved. However, it is interesting to note that the shape refinement overhead is reasonably small, although the path formulas are proportionally larger with increasing size of the shape graphs. The first three examples are chosen such that they require the same amount of shape operations in both

Program	Without joins	With joins
`simple`	0.50 s	0.57 s
`simple_backw`	0.60 s	0.62 s
`list`	3.80 s	6.92 s
`list_flag`	0.78 s	1.24 s
`alternating`	1.15 s	1.69 s
`splice`	6.98 s	9.60 s

Table 5.2: Time for verifying singly-linked list manipulation programs in seconds on a 3 GHz Intel Xeon processor; comparison of different approaches for the exploration algorithm (with and without joining nodes that agree on the predicate abstraction during the exploration of the reachability tree)

methods, to measure the overhead of lazy shape analysis compared to shape analysis, without taking advantage of the laziness.

The results of our experiments (including the C source code of our examples, the error paths, and analysis log files), as well as a pre-compiled binary of Blast 3.0, are available on the supplementary web page at `http://mtc.epfl.ch/~beyer/blast_sa`.

5.3 Comparison of Exploration Algorithms

Data-flow analysis (and shape analysis in particular) are traditionally done by fix-point algorithm, in which the lattice elements are *joined* at join points. Integrating shape analysis within the reachability analysis on the other hand does not require to perform any joins. Nevertheless, to reduce the number of explored node, an appealing idea is to join the shape graphs from abstract regions already explored for a given node that agree on the predicate abstraction (similarly to the predicated lattice approach [3]). That variation of the analysis could potentially reduce the number of explored nodes in the reachability tree. Nevertheless, experiments have shown that the cost induced by the computation of joins cannot be compensated: as shown in Table 5.2, the verification time is higher when joins are performed for all our example programs.

Another use of joins is to optimize coverage check. Under the condition that the abstract domain is relational, the coverage could be checked against the join of all shape graphs of previously visited nodes for the same location that agree on the predicate abstraction. It is foreseeable that this could result in reaching coverage earlier. Moreover, by using appropriate optimization, it would cost less than checking coverage against every node as currently implemented. Therefore, it would be worth implementing this optimization.

Chapter 6

Conclusion

We combine two approaches to software verification in order to take advantage of the strengths of both: reachability analysis using (lazy) predicate abstraction, and shape analysis.

In the context of predicate abstraction, lazy abstraction and refinement improve the scalability of the analysis by using parsimonious abstractions. Shape analysis is a very precise and flexible way of compactly representing the heap. The cost of shape analysis is generally large, though. Therefore, the main challenge is in making shape analysis more scalable. We apply lazy abstraction and refinement to shapes in order to improve scalability.

By keeping track of several different abstract domains in parallel during reachability analysis (currently predicates and shape graphs), we are able to prove the safety of a larger class of programs. Interestingly, because shapes are used only when necessary, the time for verifying programs that can be proved safe by Blast using predicate abstraction only is not impacted.

Unlike for predicate abstraction, we use a heuristic for the refinement procedure. Moreover, our refinement procedure focuses only on specific kinds of refinements (adding a points-to predicate, or adding a new node predicate). Generalizing the procedure would be useful for proving safety of programs that require richer shape abstractions (like sorting algorithms). In spite of its limitations, the current heuristic performs well on small examples.

In this work, we focus on shape analysis. It is foreseeable that lazy abstraction can be performed using other abstract domains. In particular, combining more than two abstract domains would improve the flexibility and the power of our approach. The cost of such an extension would be minimized because of the laziness of the refinement (because refinement is not performed globally, but locally). Nevertheless, our refinement heuristic does not generalize trivially to other analysis than shape analysis, and the conditions under which such a refinement can be performed are still unclear.

Bibliography

[1] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.

[2] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, LNCS 1855, pages 154–169. Springer, 2000.

[3] J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *Proc. ESEC/FSE*, pages 227–236. ACM, 2005.

[4] B. Gulavanin and S. Rajamani. Counterexample driven refinement for abstract interpretation. In *Proc. TACAS*. Springer, 2006.

[5] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.

[6] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

[7] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *Proc. SAS*, LNCS 2280, pages 280–301. Springer, 2000.

[8] K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, LNCS 2725, pages 1–13. Springer, 2003.

[9] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural functional shape analysis using local heaps. Technical Report TAU-CS-26/04, Tel-Aviv University, 2004.

[10] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. POPL*, pages 105–118. ACM, 1999.