Lazy Shape Analysis *

Dirk Beyer** Thomas A. Henzinger Grégory Théoduloz

EPFL, Switzerland

Abstract. Many software model checkers are based on predicate abstraction. If the verification goal depends on pointer structures, the approach does not work well, because it is difficult to find adequate predicate abstractions for the heap. In contrast, shape analysis, which uses graph-based heap abstractions, can provide a compact representation of recursive data structures. We integrate shape analysis into the software model checker Blast. Because shape analysis is expensive, we do not apply it globally. Instead, we ensure that, like predicates, shape graphs 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 three-valued logical structures. This approach does not only increase the precision of model checking, but it also increases the efficiency of shape analysis. We implemented the technique by extending Blast with calls to Tvla.

Keywords. Formal verification, Program analysis, Software model checking, Shape analysis, Counterexample-guided abstraction refinement, Interpolation, Predicate abstraction

1 Introduction

Counterexample-guided abstraction refinement [14,6] has dramatically increased the performance of software model checkers in the past few years [2,4]. However, being based on predicate abstractions, current model checkers are not capable of dealing effectively with recursive data structures. Shape analysis [13,5,19] is a static data-flow analysis that models the heap contents using graph-based abstractions. However, shape analysis is among the most expensive static analyses. The contribution of this paper is to show how to increase both the effectiveness of model checking and the efficiency of shape analysis by combining both techniques. By computing both predicate and shape information, we increase the precision of model checking, and thus obtain fewer false positives. The efficiency of shape analysis is improved, because expensive shape computations (such as abstract postconditions) are performed only at those program locations where the shape information is necessary to prove the verification goal. To achieve this, we apply the 'lazy abstraction' paradigm [12] to shapes.

* An abbreviated version of this paper appeared in *Proc. CAV*, LNCS 4144, pages 532–546, Springer, 2006.
** Supported in part by the MICS NCCR of the SNSF.

Dagstuhl Seminar Proceedings 06081
Software Verification: Infinite-State Model Checking and Static Program Analysis
http://drops.dagstuhl.de/opus/volltexte/2006/728

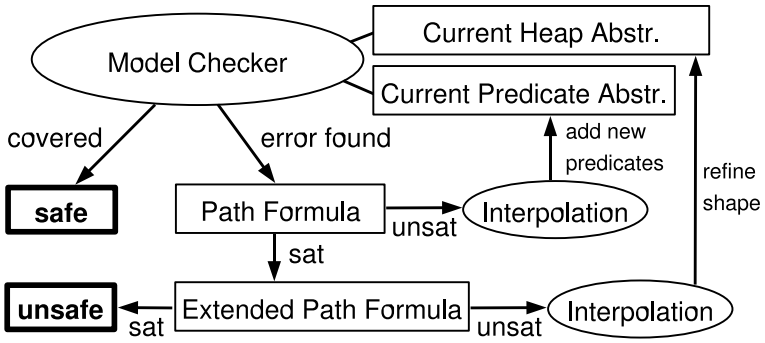


Fig. 1. Abstraction refinement with heap abstraction

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. *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. Additional predicates can be discovered automatically using Craig interpolation [18]. This method allows the pinpointing of relevant predicates to individual program locations. In this paper we show how to use interpolation-based predicate discovery to refine also the granularity of the shape analysis. Our algorithm decides, individually for each location along a spurious counterexample, which predicates and pointers to track, and how to refine the local heap abstraction, so that the infeasible error path is removed.

We refer to the predicates used in the predicate abstraction as 'nullary' predicates, because that is how they can be viewed from the shape-analysis perspective. Our interpolation engine discovers not only new nullary predicates (handled by Blast), but also new unary predicates, which are interpreted over the nodes of a shape graph. To enable the addition of richer, derived predicates (called 'instrumentation' predicates in shape analysis) during refinement, we introduce predefined shape-class generators (SCGs). Consistent with our locality principle, there is an SCG per program location. If an SCG is insufficient for proving the verification goal, then the system proceeds to a finer SCG, which adds additional shape-describing predicates to the local heap abstraction.

We implemented this algorithm in the software model checker Blast [11], using calls to Tvla [16] for shape operations. We evaluated the method by applying it to several C programs that manipulate list data structures. In these examples, the model checker needs to discover both nullary predicates (to refine the predicate abstraction) and unary predicates (to refine the heap abstraction), in order to automatically prove the program correct.

2 Review

2.1 Software Model Checking by Predicate Abstraction

Counterexample-guided abstraction refinement (CEGAR). The classical CEGAR algorithm starts with an initial (trivial) predicate abstraction, and refines the abstraction iteratively. During each iteration, it explores the states of an abstract boolean program. If the boolean program is safe, then the algorithm stops with the answer 'safe.' If an (abstract) counterexample is found, then the algorithm checks if the counterexample corresponds to a (concrete) error path in the program (which is reported as a bug), or if the counterexample is 'spurious' due to the abstraction being too coarse. In the latter case, the counterexample is analyzed to discover new predicates that need to be added to the boolean program in order to eliminate the spurious counterexample. This process is repeated until either the program is proved safe, or a bug is found [6,2]. It is possible that the process does not terminate, or that no suitable new predicates are discovered even though the counterexample is spurious.

Lazy abstraction refinement. The classical version of the abstract-check-refine loop has two drawbacks: first, it is not necessary to represent in the boolean abstraction the part of the state space that is not reachable, and second, it is not necessary to refine the portions of the abstract program that have already been proved safe. Lazy abstraction refinement integrates the steps of the abstract-check-refine loop into an on-the-fly analysis that refines the predicate abstraction locally. Instead of repeatedly building and exploring an abstract boolean program, the lazy algorithm builds an ART. At each node of the ART, the lazy algorithm adds necessary predicates on demand, by refining the abstraction only at locations that occur on a spurious counterexample. As a result, the final abstraction (predicate set) differs from location to location [12].

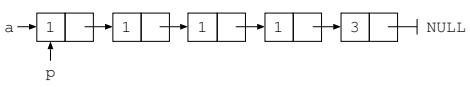
Craig interpolation. The crucial measure for the efficiency of the analysis is the number of predicates in the abstraction. To keep the number of predicates per location as small as possible, interpolation-based predicate discovery can be used to produce for each program location the predicates that are needed to eliminate an infeasible error path in the ART. Given an abstract error path, we construct a path formula (PF) such that if the PF is unsatisfiable, then the error path is infeasible. An unsatisfiable PF can be cut, at each location on the path, into two formulas: a prefix formula that leads the program from the initial location to the cut location, and a postfix formula that leads the program from the cut location to the error location. From a Craig interpolant of the two formulas we can extract a suitable set of predicates to be added at the cut location [11,18].

2.2 Shape Analysis by Three-Valued Logic

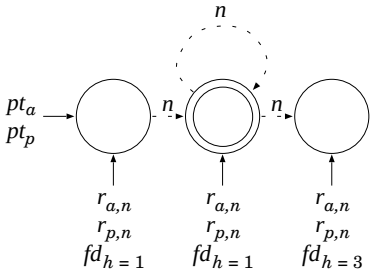
Shape analysis is a static analysis that represents unbounded instances of recursive data structures on the heap by finite structures, called 'shape graphs.' Following the framework of [19,16], we represent shape graphs as three-valued logical structures.

```
1 typedef struct node {
2   int h;  struct node *n;
3 } *List;
4 void foo(int flag) {
5   List a = (List) malloc(...);
6   if (a == NULL) exit(1);
7   List p = a;
8   while (random()) {
9     if (flag)  p->h = 1;
10    else       p->h = 2;
11    p->n = (List) malloc(...);
12    if (p->n == NULL) exit(1);
13    p = p->n;  }
14  p->h = 3;
15  p = a;  /* Now check it. */
16  if (flag)  while (p->h == 1)  p = p->n;
17  else       while (p->h == 2)  p = p->n;
18  assert(p->h == 3);
19 }
```

(a) Example C program



(b) Concrete list



(c) Shape of the concrete list

Fig. 2. Example program and two list representations

Figure 2(b) shows an instance of a list structure consisting of five 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(c) shows a shape graph that represents all list instances such that the pointers `a` and `p` point to the first list element, all elements have data value 1, except the last element, which has data value 3. The concrete list in Fig. 2(b) is an instance of this shape graph. The shape graph is represented by predicates: the unary predicates pt_a, pt_p, $fd_{h=1}$, $fd_{h=3}$, sm, $r_{a,n}$, and $r_{p,n}$, and the binary predicate n. All predicates are interpreted over nodes of the shape graph. The predicate $pt_a(v)$ is true if the pointer variable `a` points to node v (same for pt_p); the predicate $n(v, u)$ is true if the next pointer of node v points to node u; the predicate $fd_{h=1}(v)$ is true if the field `h` of node v satisfies the assertion `h=1` (same for $fd_{h=3}$); the predicate $r_{a,n}(v)$ is true if node v is reachable from pointer `a` via the next-pointer relation (same for $r_{p,n}$); and the predicate $sm(v)$ has the value 1/2 if v is a summary node, and the value *false* if v represents a single list element. A summary node (drawn as double-circled) represents one or more list elements. The next pointer of a list element that is abstracted by the second node in Fig. 2(c) may point to itself or to the third node (the value 1/2 of a predicate is indicated by a dotted edge). The reachability predicates $r_{a,n}$ and $r_{p,n}$ are defined in terms of the other predicates; they are called 'instrumentation' predicates. All other predicates are 'core' predicates.

3 Preview

Lazy CEGAR with shapes. We define a lazy CEGAR algorithm for abstractions that consist of a predicate abstraction and a heap abstraction (cf. Fig. 1). Moreover, following the lazy abstraction paradigm, both abstractions are refined

locally. The initial predicate abstraction is the single predicate *true*, and the initial heap abstraction is the trivial shape graph, which represents every heap. With each program operation, we update both abstractions independently. During lazy abstraction refinement, if the PF is unsatisfiable, then the spurious counterexample is due to the predicate abstraction, and the interpolation procedure discovers new predicates that are added to the predicate abstraction. In this case, the heap abstraction is not changed. However, if the PF is satisfiable, this does not necessarily mean that the error path is feasible. In this case, we construct a more precise extended path formula (EPF), which takes into account also information about the heap. If the EPF is unsatisfiable, then the error path is guaranteed to be infeasible. We apply the interpolation procedure to the EPF, and use the interpolants to decide how to refine the heap abstraction at the cut locations. For example, from an interpolant of the form `p->h=3`, we extract the new predicates pt_p and $fd_{h=3}$ to refine the shape graph.

Example. The function in Fig. 2(a) first builds a list that contains a sequence of data values in $\{1, 2\}$ —depending on the variable `flag`— and ends with data value 3. Then the function verifies that property of the list. Pure predicate abstraction discovers only the nullary predicate np_{flag}, which is insufficient for proving the program safe. The combination of predicate abstraction and heap abstraction tracks both predicate and shape information simultaneously, and automatically discovers the necessary nullary and unary predicates to refine both abstractions. The first infeasible error path that our system 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, $\langle 3 \rangle$. Pure predicate abstraction would give a false positive, because the PF is satisfiable. However, the EPF is unsatisfiable, and from the interpolant `p->h=3` we extract the pointer `p` and the field assertion `h=3`. Furthermore, alias analysis indicates that we also need to track the pointer `a`, which may be aliased to `p`. Therefore we locally add the three unary predicates pt_p, pt_a, and $fd_{h=3}$ to the heap abstraction, which removes the infeasible error path.

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 now represents the sequence $\langle 2, 3 \rangle$. The abstract state associated with the program location before the assertion is represented by the nullary predicate *true* and the shape graph of Fig. 4(a). The conjuncts of the EPF that show the infeasibility of this error path are given in Fig. 3 (the number annotated to an lvalue in a PF corresponds to the number of the operation that has written this value). The interpolant is `p->h=2`, and thus we add the field predicate $fd_{h=2}$ to the heap abstraction. When BLAST explores this path again after the refinement, the shape graph in Fig. 4(b) is computed.

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 $\langle 1, 3 \rangle$. As the predicate abstraction does not track the predicate `flag`, this leads to the infeasible situation that in the first while loop the predicate is assumed to be *true*,

Operation	Constraint
1 : `a=malloc()`	*true*
2 : *assume*(`a!=0`)	$\langle a,1\rangle \neq 0$
3 : `p=a`	$\langle p,3\rangle = \langle a,1\rangle \wedge \langle\langle p,3\rangle\texttt{->}h,3\rangle = \langle\langle a,1\rangle\texttt{->}h,1\rangle$ $\wedge\ \langle\langle p,3\rangle\texttt{->}n,3\rangle = \langle\langle a,1\rangle\texttt{->}n,1\rangle$
4 : *assume*(`flag==0`)	$\langle flag,0\rangle = 0$
5 : `p->h=2`	$\langle\langle p,3\rangle\texttt{->}h,5\rangle = 2 \wedge \langle\langle a,1\rangle\texttt{->}h,5\rangle = 2$
6 : `p->n=malloc()`	*omitted*
7 : *assume*(`p->n!=0`)	*omitted*
8 : `p=p->n`	*omitted*
9 : `p->h=3`	*omitted*
10 : `p=a`	$\langle p,10\rangle = \langle a,1\rangle \wedge \langle\langle p,10\rangle\texttt{->}h,10\rangle = \langle\langle a,1\rangle\texttt{->}h,5\rangle$ $\wedge\ \langle\langle p,10\rangle\texttt{->}n,10\rangle = \langle\langle a,1\rangle\texttt{->}n,1\rangle$
11 : *assume*(`flag==0`)	$\langle flag,0\rangle = 0$
12 : *assume*(`p->h!=2`)	$\langle\langle p,10\rangle\texttt{->}h,10\rangle \neq 2$
13 : *assume*(`p->h!=3`)	$\langle\langle p,10\rangle\texttt{->}h,10\rangle \neq 3$
14 : `ERROR`	

Fig. 3. Extended path formula for the second infeasible error path

(a) Shape before refinement (violating) (b) Shape after refinement (not violating)

Fig. 4. Shape graphs before and after the second refinement at program line 18

and in the second part of the program it is assumed to be *false*. The interpolant for the unsatisfiable PF is `flag`, and we add the nullary predicate np_{flag} to the predicate abstraction. The resulting 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 discover the field predicate $fd_{h=1}$ from the interpolant `p->h=1` for the unsatisfiable EPF, and add it to the heap abstraction.

The last iteration unfolds the remaining states of the ART or marks them as covered. The final ART represents a safety certificate (proof of correctness). The example does not illustrate the 'laziness' of our approach: if the example is only one function of many in a large program, then the generated predicates and shape graphs are tracked only locally within the given function. Similarly, if the program contained a second list that is created but never checked, then the analysis would not track the shape of that list, because the interpolants yield only predicates that are necessary for eliminating the infeasible error paths. Also, the example uses only one of the two ways in which the heap abstraction can be refined: it does not require any instrumentation predicates. To introduce derived predicates, such as reachability, we will add shape-class generators to the heap abstraction; these are not discovered automatically, but need to be predefined.

4 Lazy Abstraction Refinement of Shapes

In the following three subsections, we give the details of our algorithm. First, we explain how we build the ART with abstract states that include both nullary predicates and shape graphs. Second, we explain how we check whether an abstract error path is feasible (i.e., corresponds to a concrete error path). Third, we explain how we refine the predicate and heap abstractions.

4.1 Combining Predicate and Heap Abstractions

Predicate abstraction. A *nullary predicate* is a predicate over program variables. We write $np_{x=3}$ for the nullary predicate asserting that the value of the program variable `x` is 3. We denote the set of nullary predicates by $\mathcal{P}$. A *predicate abstraction* for a program is a function $\Pi : L \to 2^{\mathcal{P}}$ that maps each program location in $L = \{pc_1, \ldots, pc_k\}$ to a set of nullary predicates. We follow the definitions of [11]. For a formula φ over program variables, the abstraction w.r.t. a set $P \subseteq \mathcal{P}$ of nullary predicates is the strongest boolean combination φ' of predicates in P such that φ implies φ'. The semantics of a program path is defined in terms of the strongest-postcondition operator: if the formula φ represents a set of states, and op is an operation, then the formula $\mathsf{SP}.\varphi.\mathsf{op}$ represents the set of successor states. We extend SP to program paths in the natural way. A path t is SP-*infeasible* if $\mathsf{SP}.true.t$ is unsatisfiable. To check if a given path is feasible, we construct a path formula (PF), which is the conjunction of several constraints, one per operation on the path, such that the path is SP-infeasible if the PF is unsatisfiable. For a path $(\mathsf{op}_m : pc_m); \ldots; (\mathsf{op}_n : pc_n)$, the *abstract semantics* SP_Π is the Π-abstraction of the concrete semantics SP, that is, the formula $\mathsf{SP}_\Pi.\varphi.\mathsf{op}_i$ is the abstraction w.r.t. $\Pi(pc_i)$ of the formula $\mathsf{SP}.\varphi.\mathsf{op}_i$.

Shape classes. The precision of heap abstractions is defined by shape classes. Following [19], a *shape class* $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$ consists of three sets of predicates over node variables: (1) a set P_{core} of core predicates, (2) a set P_{instr} of instrumentation predicates with $P_{core} \cap P_{instr} = \emptyset$, where each instrumentation predicate $p \in P_{instr}$ has an associated defining formula φ^p over the core predicates, and (3) a set $P_{abs} \subseteq P_{core} \cup P_{instr}$ of abstraction predicates. We denote the set of shape classes by $\mathcal{S}$. A *heap abstraction* for a program is a function $\Psi : L \to 2^{\mathcal{S}}$ that maps each program location to a set of shape classes (different shape classes can be used to simultaneously track different data structures).

The set of core predicates must contain the special unary predicate *sm*, which has the value *false* for normal nodes and 1/2 for summary nodes. Moreover, we distinguish two special subsets of core predicates: the set $P_{pt} \subseteq P_{core}$ of points-to predicates, and the set $P_{fd} \subseteq P_{core}$ of field predicates. A *points-to predicate* $pt_x(v)$ is a unary predicate that indicates if a pointer variable `x` points to node v. A *field predicate* $fd_\phi(v)$ is a unary predicate that indicates if a field assertion ϕ holds for node v. Each field assertion has a boolean value over the fields of a structure element. Therefore, field predicates represent the data content of a structure, rather than the shape of the structure. A shape class $\mathbb{S}$ *refines* a shape class $\mathbb{S}'$, written $\mathbb{S} \preccurlyeq \mathbb{S}'$, if (1) $P'_{core} \subseteq P_{core}$,

(2) $P'_{instr} \subseteq P_{instr}$, and (3) $P'_{abs} \subseteq P_{abs}$. The union of $\mathbb{S}$ and $\mathbb{S}'$ is the shape class $(P_{core} \cup P'_{core}, P_{instr} \cup P'_{instr}, P_{abs} \cup P'_{abs})$ (assuming $P_{core} \cap P'_{instr} = \emptyset$ and $P_{instr} \cap P'_{core} = \emptyset$).

Shape graphs. The abstract state of the heap is defined as a set of shape regions. A *shape region* $(\mathbb{S}, G)$ consists of a shape class $\mathbb{S}$ and a set G of shape graphs for $\mathbb{S}$. A *shape graph* $g = (V, val)$ for a shape class $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$ consists of a set V of nodes and a valuation val in three-valued logic of the predicates over V: for a predicate $p \in P_{core} \cup P_{instr}$ of arity n, $val(p) : V^n \to \{0, 1, 1/2\}$. Fig. 2(c) shows an example of a shape graph. Let $\mathbb{S}$ and $\mathbb{S}'$ be two shape classes such that $\mathbb{S} \preccurlyeq \mathbb{S}'$. A shape graph g' for $\mathbb{S}'$ can be extended to a shape graph $g = E_{\mathbb{S}' \triangleright \mathbb{S}}(g')$ for $\mathbb{S}$ such that the set of nodes is unchanged (i.e., $V = V'$), and for each predicate $p \in (P_{core} \cup P_{instr}) \setminus (P'_{core} \cup P'_{instr})$, the value of p is $1/2$ everywhere. We extend the operator E to sets of shape graphs in the natural way. A shape region $(\mathbb{S}, G)$ is *covered* by a shape region $(\mathbb{S}', G')$ if $E_{\mathbb{S} \triangleright (\mathbb{S} \cup \mathbb{S}')}(G) \subseteq E_{\mathbb{S}' \triangleright (\mathbb{S} \cup \mathbb{S}')}(G')$. Consider a program path $(\mathsf{op}_m : pc_m); \ldots; (\mathsf{op}_n : pc_n)$ and a heap abstraction Ψ. The *abstract semantics* SP_Ψ is the Ψ-abstraction of the concrete semantics SP, that is, the formula $\mathsf{SP}_\Psi.(\mathbb{S}, G).\mathsf{op}_i$ is the abstraction w.r.t. $\Psi(pc_i)$ of the formula $\mathsf{SP}.\varphi.\mathsf{op}_i$. The operator SP_Ψ is defined by $\mathsf{SP}_\Psi.(\mathbb{S}, G).\mathsf{op}_i = (\mathbb{S}, [\![\mathsf{op}_i]\!](G))$, where $[\![\cdot]\!]$ is defined as in TVLA [16]. To compute SP_Ψ, we use TVLA's *focus* and *coerce* functions to transform a set of shape graphs. We extend the notion of being covered and the operator SP_Ψ to sets of shape regions in the natural way.

ART construction. The *abstraction* (Π, Ψ) of a program is a pair consisting of a predicate abstraction Π and a heap abstraction Ψ. Given (Π, Ψ), we construct an ART following [12], but with each vertex of the ART we store not only a program location $pc \in L$, a call stack, and a subset of the (nullary) predicates in $\Pi(pc)$, but also a set of shape regions, one for each shape class in $\Psi(pc)$. Successor vertices in the ART are computed using the SP_Π and SP_Ψ operators independently on the two parts (nullary predicates and shape regions) of the abstract state. We stop expanding the ART at a vertex if (1) both the set of nullary predicates, and the set of shape regions, are covered by some other vertex; or (2) either the predicate set, or the shape set, represents the empty set of concrete states. More sophisticated termination criteria are possible, of course.

4.2 Extracting Interpolants from Extended Path Formulas

Programs. Our formalization of programs is similar to [11]. A program is represented by a set of control flow automata; a path t of length n is a sequence $(\mathsf{op}_1 : pc_1); \ldots; (\mathsf{op}_n : pc_n)$ of operations, which can be either statements or assume predicates. In 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 pointer to structures. We restrict the lvalues that can occur in a program to *ident* and *ident*->*field*, where *ident* denotes a variable identifier and *field* denotes the name of a structure field. The function F maps an lvalue to the set of fields of the structure pointed to by

the lvalue if the lvalue has a pointer type, and to the empty set if the lvalue has an integer type. The operations within a program are limited to the ones listed in the first column of Fig. 5. The expressions that can occur in statements are side-effect free C expressions of linear arithmetic, without pointer dereferences.

Extended path formulas. The technique for building PFs from [11] cannot be reused directly, because it does not refer to recursive data structures. However, since the number of memory cells possibly involved in a PF is bounded, we can produce a finite, sound formula called extended PF (EPF). The address of each heap cell that is accessed on a path must have been previously assigned to a pointer variable (because we consider a restricted set of possible lvalues). To refer to these addresses in the EPF, we use SSA-like renamed lvalues. An *lvalue constant* is either $\langle \mathit{ident}, l \rangle$ (a variable constant), or $\langle \langle \mathit{ident}, l \rangle \text{->} \mathit{field}, l' \rangle$ with position labels $l, l' \in [0..n]$ and $l' \geq l$. An *annotated lvalue* is either *ident*, or $\langle \mathit{ident}, l \rangle \text{->} \mathit{field}$. The labels l and l' identify the positions on the path where the annotated values may have been modified. An annotated-lvalue map θ is a function from annotated lvalues to labels. The *lvalue-renaming function* $\mathsf{sub}.\theta.v$ is defined by $\mathsf{sub}.\theta.s = \langle s, \theta(s) \rangle$ and $\mathsf{sub}.\theta.(s\text{->}f) = \langle (\mathsf{sub}.\theta.s)\text{->}f, \theta((\mathsf{sub}.\theta.s)\text{->}f) \rangle$, where s is a variable and f is a field.

To simplify the EPF using alias information, the function may maps a label and an lvalue constant to the set of variable constants that may have the same value (i.e., $\langle s, l_s \rangle \in \mathsf{may}.l.c$ if, after the l-th operation of the path, the value of c may be equal to the value of s_1 after the l_1-th operation of the path). The function may is not essential: it is used only to reduce the size of the EPF by taking into account information that two pointers are guaranteed not to be equal.

The function FineCon maps a pair (θ, Γ) consisting of an annotated-lvalue map θ and a constraint map Γ from position labels to first-order logic formulas over lvalue constants, and an operation op_i, to a pair (θ', Γ') consisting of a new annotated-lvalue map and a new constraint map. On the given path, we compute recursively the result of FineCon by computing $(\theta_l, \Gamma_l) = \mathsf{FineCon}.(\theta_{l\text{-}1}, \Gamma_{l\text{-}1}).\mathsf{op}_l$, where l is the position label of op_l on the path. The map θ_0 is the constant function 0, and Γ_0 is the constant function $\emptyset$. The map θ_l differs from $\theta_{l\text{-}1}$ only on the annotated lvalues that may be modified by op_l, which are mapped to l by θ_l. The map Γ_l results from extending $\Gamma_{l\text{-}1}$ by mapping l to the constraint derived from op_l. We derive constraints from operations similarly to [11]. An extension is necessary for assignments to pointers: we cannot 'unroll' a recursive data structure and refer to all reachable memory cells, because this would yield an infinite formula. Additionally, we need to add aliasing constraints when several lvalue constants may point to the same memory cell. The formal definition of the function FineCon is given in Fig. 5. The EPF of the path is obtained by taking the conjunction of all formulas in the final constraint map. The EPF is unsatisfiable iff the path is SP-infeasible.

The definition of FineCon uses the following two functions. The function eqvar returns a constraint that expresses the equality of two variables by considering their fields (if any):

Op. op_l	New map θ' and Alloc'	Constraint $\Gamma'(l)$
$s = e$	$\theta'(s) = l$	$\mathsf{sub}.\theta'.s = \mathsf{sub}.\theta.e$
$s_1 = s_2$	$\theta'(s_1) = l$ $\forall f \in \mathsf{F}(s_1) : \theta'(\langle s_1, l\rangle \mathsf{->} f) = l$	$\mathsf{eqvar}.(s_1, \theta').(s_2, \theta)$
$s_1 = s_2 \mathsf{->} f$	$\theta'(s_1) = l$ $\forall f \in \mathsf{F}(s_1) : \theta'(\langle s_1, l\rangle \mathsf{->} f) = l$	$\mathsf{sub}.\theta'.s_1 = \mathsf{sub}.\theta.(s_2 \mathsf{->} f)$ $\wedge \bigwedge_{c \in \mathsf{may}.(l\text{-}1).(\mathsf{sub}.\theta.(s_2 \mathsf{->} f))} \left(\begin{array}{l}\mathsf{sub}.\theta.(s_2 \mathsf{->} f) = c \\ \Rightarrow \mathsf{eqvar}.(s_1, \theta').(c, \theta)\end{array}\right)$
$s_1 \mathsf{->} f = s_2$	$\theta'(\langle s_1, \theta(s_1)\rangle \mathsf{->} f) = l$ $\forall c \in \mathsf{may}.(l\text{-}1).\langle s_1, \theta(s_1)\rangle \mathsf{->} f, l\rangle$: $\quad \forall f \in \mathsf{F}(c) : \theta'(\langle c, l\rangle \mathsf{->} f) = l$ $\forall c \in \mathsf{may}.(l\text{-}1).\langle s_1, \theta(s_1)\rangle$: $\quad \theta'(c \mathsf{->} f) = l$	$\mathsf{sub}.\theta'.(s_1 \mathsf{->} f) = \mathsf{sub}.\theta.s_2$ $\wedge \bigwedge_{c \in \mathsf{may}.(l\text{-}1).(\mathsf{sub}.\theta'.(s_1 \mathsf{->} f))} \left(\begin{array}{l}\mathsf{ite}.(c = \mathsf{sub}.\theta'.(s_1 \mathsf{->} f)) \\ .(\mathsf{eqvar}.(c, \theta').(s_2, \theta)) \\ .(\mathsf{eqvar}.(c, \theta').(c, \theta))\end{array}\right)$ $\wedge \bigwedge_{c \in \mathsf{may}.(l\text{-}1).(\mathsf{sub}.\theta'.s_1)} \left(\begin{array}{l}\mathsf{ite}.(c = \mathsf{sub}.\theta'.s_1) \\ .(\mathsf{sub}.\theta'.(c \mathsf{->} f) = \mathsf{sub}.\theta.s_2) \\ .(\mathsf{sub}.\theta'.(c \mathsf{->} f) = \mathsf{sub}.\theta.(c \mathsf{->} f))\end{array}\right)$
$s = alloc()$	$\theta'(s) = l$ $\forall f \in \mathsf{F}(s) : \ \theta'(\langle s, l\rangle \mathsf{->} f) = l$ $\mathsf{Alloc}' = \mathsf{Alloc} \cup \{\langle s, l\rangle\}$	$\bigwedge_{a \in \mathsf{Alloc}} (\langle s, l\rangle \neq a)$
$assume(p)$		$\mathsf{clos}^*.\theta.true.p$

Fig. 5. Definition of $\mathsf{FineCon}$ for each operation: $(\theta', \Gamma') = \mathsf{FineCon}.(\theta, \Gamma).l.\mathsf{op}_l$

$$\mathsf{eqvar}.(s_1, \theta_1).(s_2, \theta_2) \ = \ (\mathsf{sub}.\theta_1.s_1 = \mathsf{sub}.\theta_2.s_2) \wedge \bigwedge_{f \in \mathsf{F}(s_1)} (\mathsf{sub}.\theta_1.(s_1 \mathsf{->} f) = \mathsf{sub}.\theta_2.(s_2 \mathsf{->} f))$$

The function $\mathsf{clos}^*.\theta.b.p$ returns, given an assume predicate p, the predicate that results from replacing all equalities $x_1 = x_2$ occurring positively (or negatively, depending on the value of the boolean value b) by $\mathsf{eqvar}.(x_1, \theta).(x_2, \theta)$:

$$\mathsf{clos}^*.\theta.b.p = \begin{cases} (\mathsf{clos}^*.\theta.b.p_1) \ \mathsf{op} \ (\mathsf{clos}^*.\theta.b.p_1) & \text{if } p \equiv (p_1 \ \mathsf{op} \ p_2) \\ \neg(\mathsf{clos}^*.\theta.\neg b.p_1) & \text{if } p \equiv (\neg p_1) \\ \mathsf{eqvar}.(x_1, \theta).(x_2, \theta) & \text{if } p \equiv (x_1 = x_2) \text{ and } b \equiv true \\ \mathsf{sub}.\theta p & \text{otherwise} \end{cases}$$

Interpolation. We compute the interpolants using the algorithm *Extract* from [11]. We parametrize the algorithm either with the function Con [11] for PFs (written *Extract*[Con]), or with the new function $\mathsf{FineCon}$ for EPFs (written *Extract*[$\mathsf{FineCon}$]). The algorithm *Extract* takes as input a program path t and returns a function $\hat{\Pi}$ from position labels (i.e., locations on the path) to sets of nullary predicates. In [11] it was shown that, for a weaker programming language (without recursive data structures), the path t is SP-infeasible iff t is $\mathsf{SP}_{\hat{\Pi}}$-infeasible for $\hat{\Pi} = \mathit{Extract}[\mathsf{Con}](t)$. We can prove the analogous statement

for our richer programming language: the path t is SP-infeasible iff t is $\mathsf{SP}_{\hat{\Pi}}$-infeasible for $\hat{\Pi} = \mathit{Extract}[\mathsf{FineCon}](t)$ [3]. Therefore, our method is sound, that is, we do not report safety when a bug exists.

4.3 Shape-Class Refinement Based on Interpolants

Our refinement procedure first tries to refine the predicate abstraction, by locally adding to the predicate abstraction nullary predicates from the interpolants. If the algorithm does not find new predicates to refine the predicate abstraction, then it tries to refine the heap abstraction, by locally refining the shape classes. In order to specify heap abstractions, we introduce the following data structures.

Tracking definitions and shape-class generators. A tracking definition represents the pointers and field predicates that we track for analyzing the heap. A *tracking definition* $D = (T, T_s, \Phi)$ consists of (1) a set T of *tracked pointers*, which is the set of variables that may be pointing to some node in a shape graph; (2) a set $T_s \subseteq T$ of *separating pointers*, which is the set of variables for which we want the corresponding predicates (e.g., points-to, reachability) to be abstraction predicates; and (3) a set Φ of field assertions. A tracking definition $D = (T, T_s, \Phi)$ *refines* a tracking definition $D' = (T', T'_s, \Phi')$, if $T' \subseteq T$, $T'_s \subseteq T_s$ and $\Phi' \subseteq \Phi$. We denote the set of all tracking definitions by $\mathcal{D}$.

A *shape-class generator* (SCG) is a function $m : \mathcal{D} \to \mathcal{S}$ that takes as input a tracking definition and returns a shape class, which consists of core predicates, instrumentation predicates, and abstraction predicates. The tracking definition provides information about which pointers and which field predicates need to be tracked by the program analysis; it is the SCG that determines which predicates are actually added to the shape class. Useful SCGs produce at least the summary predicate sm, a points-to predicate pt_x for each pointer $x \in T$ in the tracking definition, and a field predicate fd_ϕ for each field assertion $\phi \in \Phi$ in the tracking definition. While the pointers and field assertions in the tracking definition are discovered by interpolation (see below), predicates other than sm, pt, and fd predicates need to be added by defining appropriate SCGs.[1] An SCG m *refines* an SCG m' if $m(D) \preccurlyeq m'(D)$ for all tracking definitions D. We require that the set of SCGs contains at least the greatest element m_0, which is a constant function that generates for each tracking definition the shape class $(\emptyset, \emptyset, \emptyset)$. Furthermore, we require each SCG to be monotonic: given an SCG m and two tracking definitions D and D', if $D \preccurlyeq D'$, then $m(D) \preccurlyeq m(D')$.

A *shape type* $\mathbb{T} = (\sigma, m, D)$ consists of a C structure type σ, an SCG m, and a tracking definition D. For example, consider the C type `struct node {int data; struct node *next;};` and the tracking definition $D = (\{p, q\}, \{p\}, \{data = 0\})$. To form a shape type for a singly-linked list, we can choose an SCG that takes a tracking definition $D = (T, T_s, \Phi)$ and produces a shape class $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$ with the following components: the set P_{core} of core predicates contains the default unary predicate sm for distinguishing

[1] BLAST uses a predefined set of SCGs, to limit the space of shape classes our algorithm considers. Technically, in BLAST each SCG is given as a code module.

summary nodes, a binary predicate *next* for representing links between nodes in the list, for each variable in T a unary points-to predicate, and a unary field predicate for each assertion in Φ. The set P_{instr} of instrumentation predicates contains for each variable in T a reachability predicate. The set P_{abs} of abstraction predicates contains all core and instrumentation predicates about separating pointers from T_s. More precise shape types for singly-linked lists can be defined by providing a generator that adds more instrumentation predicates.

A *heap-abstraction specification* is a function $\hat{\Psi}$ that assigns to each program location a set of shape types. The specification $\hat{\Psi}$ defines a heap abstraction in the following way: a pair $(l, \{\mathbb{T}_1, \ldots, \mathbb{T}_k\}) \in \hat{\Psi}$ yields a pair $(l, \{\mathbb{S}_1, \ldots, \mathbb{S}_k\}) \in \Psi$ with $\mathbb{S}_i = \mathbb{T}_i.m(\mathbb{T}_i.D)$ for all $1 \leq i \leq k$. Technically we do not store the heap abstraction Ψ in our system, but only its specification $\hat{\Psi}$. Whenever a shape class is needed, the algorithm looks it up by applying the current shape type's SCG to the shape type's tracking definition. The tracking definition contains information about which pointers and field assertions to track on a syntactic level. Since SCGs are monotonic, shape types can be refined in two different ways: either we refine the shape type's tracking definition, or we refine the shape type's SCG. In both cases, the produced shape class is guaranteed to be finer.

Refinement algorithm. In the abstract-check-refine loop, predicate abstraction starts with empty set of predicates, and heap abstraction starts with empty shape classes at all program locations. The input to the refinement algorithm is a path t to an error location, which is feasible under the current abstraction (Π, Ψ) (i.e., t is contained in the current ART). Following [11], the algorithm first checks the PF of t for satisfiability. If the PF is unsatisfiable, then the predicate abstraction Π is refined by adding interpolants from $\mathit{Extract}[\mathsf{Con}](t)$ to the locations on the path t. Otherwise, we check the EPF of t for satisfiability. If the EPF is satisfiable, then we report a program bug. Otherwise, we compute new interpolants using $\mathit{Extract}[\mathsf{FineCon}](t)$, and consider each location pc on the path t separately. We either refine the tracking definition of pc in Step 1, or the SCG of pc in Step 2. The algorithm always tries Step 1 first, and only if neither new pointers nor new field assertions are discovered from the corresponding interpolant, it tries Step 2. This interpolation-based analysis identifies the program locations that require more precision to remove the error path from the ART, and we refine the heap abstraction only for those locations.

Step 1: [refine the tracking definition of a location] For every pointer variable `p` that occurs in the interpolant (e.g., `p->h=3`), if it matches the C type of the shape type, then we refine the tracking definition of the shape type as follows. We add `p` to the set of tracked pointers and to the set of separating pointers, and we close the set of tracked pointers under aliasing, by adding also all pointer variables that may be pointing to the newly tracked data structure. Thus the quality of the refinement depends on the quality of the available may-alias information: imprecise information may cause some *pt* predicates to be added unnecessarily.[2] Moreover, if the pointer `p` is dereferenced in the interpolant, then we add the

[2] BLAST currently uses a flow-insensitive may-alias algorithm in which all cells allocated at the same site are represented by one abstract cell.

Algorithm 1 *AbstractionRefinement*$(t, (\Pi, \Psi), M)$

Input: a program path $t = (\mathsf{op}_1 : pc_1); \ldots; (\mathsf{op}_n : pc_n)$, an abstraction (Π, Ψ), a set M of shape-class generators
Output: an abstraction (Π', Ψ')
Variables: a predicate abstraction $\hat{\Pi}$
$\Pi' := \Pi$; $\Psi' := \Psi$;
$(\cdot, \Gamma) := \mathsf{Con}.(\theta_0, \Gamma_0).t$; // Non-extended constraints.
if $\bigwedge_{1 \le i \le n} \Gamma.i \vdash \mathit{false}$ **then**
  $\Pi' := \Pi \sqcup \mathit{Extract}[\mathsf{Con}](t)$;
else
  $(\cdot, \Gamma) := \mathsf{FineCon}.(\theta_0, \Gamma_0).t$; // Extended constraints.
  if $\bigwedge_{1 \le i \le n} \Gamma.i \vdash \mathit{false}$ **then**
    $\hat{\Pi} := \mathit{Extract}[\mathsf{FineCon}](t)$;
    for $i := 1$ to n **do**
      let $\Psi.i = (\sigma, m, D)$ and $\Psi'.i = (\sigma', m', D')$;
      for each atom $\phi \in \hat{\Pi}.i$ **do**
        if pointer p occurs in ϕ, and $\mathit{type}(p)$ matches σ' **then**
          $D'.T := D'.T \cup \{p\} \cup \mathsf{alias}(p)$;
          $D'.T_s := D'.T_s \cup \{p\}$;
          if pointer p is dereferenced in ϕ **then**
            construct the field assertion ϕ' corresponding to ϕ;
            $D'.P := D'.P \cup \{\phi'\}$;
      if $D = D'$ **then**
        let $\hat{M}$ be the set of coarsest generators $\hat{m} \in M$ such that $\hat{m} \neq m$, $\hat{m} \preccurlyeq m$;
        if $\hat{M} = \emptyset$ **then**
          throw exception "No refinement found";
        else
          choose $\hat{m} \in \hat{M}$; $m' := \hat{m}$;
  else
    print "Bug found."; **stop**;
return (Π', Ψ')

corresponding field assertion (e.g., `h=3`) to the set of tracked field assertions. The same SCG now produces a finer shape class for the refined tracking definition.

Step 2: [refine the SCG of a location] The choice of a finer SCG from the predefined set of SCGs is guided by a refinement relation over the SCGs, which can be used to encode various heuristics that analyze the abstract error path. The finer SCG produces for the same tracking definition a finer shape class, by adding new core, instrumentation, and/or abstraction predicates (e.g., sharing, reachability, cyclicity). If such a finer SCG cannot be found, due to the limitation to a predefined set of SCGs, then the algorithm reports that refinement has not succeeded and terminates. The predefined set of SCGs can be extended in a very flexible way to arbitrary data structures by developers of the model checker, and by experienced users. Given an infeasible program path t and a finite set M of SCGs, it can be proved that the iterative application of the refinement algorithm eventually produces an abstraction (Π', Ψ') that is able to remove t from the ART, provided that M contains an appropriate SCG [3]. Completeness, in this sense, hinges on the use of a sufficiently rich set of SCGs.

5 Evaluation

Implementation. The algorithm presented in this paper is implemented in Blast 3.0, which integrates Tvla for shape transformations and the foci library [18] of Blast 2.0 for predicate interpolation. Blast's abstract states, which were triples consisting of program counter, call stack, and (nullary) predicate set, are extended by a set of shape regions, one for each tracked shape class. Tvla (written in Java) is integrated into Blast (written in OCaml) as a particular implementation of a shape-analysis module and can be replaced by other shape-analysis tools. We use a simple home-brewed may-alias analysis, but this module can also be changed.

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 traversed checking that it contains only the same number.

Table 1 reports the results of our experiments. None of the programs was successfully verified by Blast's predicate abstraction without shape analysis (only nullary predicates): 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-based heap analysis (without nullary predicates for tracking control flow, and with tracking maximal shape information everywhere),

Table 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)

Program	CFA nodes	LOC	Pure pred. abstr.	Pure heap analysis	PA & SA
`simple`	26	44	FP 0.16 s (0)	0.48 s	0.51 s (1)
`simple_backw`	19	39	FP 0.36 s (4)	0.43 s	0.58 s (5)
`list`	34	54	FP 0.15 s (0)	3.74 s	4.63 s (3)
`list_flag`	35	62	FP 0.15 s (0)	FP 0.26 s	1.18 s (4)
`alternating`	30	58	FP 0.20 s (1)	FP 0.26 s	1.77 s (5)
`splice`	42	84	FP 0.68 s (3)	FP 0.66 s	6.10 s (7)

but for the other three it fails due to missing control-flow sensitivity (column five). The model checker BLAST with *lazy shape analysis* discovers automatically all necessary predicates to prove each of the example programs safe (last column). Our examples can also be proved by TVLA, giving as input the abstraction that our system constructs automatically.

The experiments of the first three programs show that the overhead for the automatic discovery of relevant points-to and field predicates in BLAST with integrated shape analysis does not significantly increase the run-time of the analysis, compared to shape analysis when given the final abstraction (without taking advantage of the laziness). 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 the shape analysis they require is more involved.

Since our system uses different abstractions at different program locations, while TVLA performs a global analysis, there are examples on which our system significantly outperforms TVLA. Also, BLAST uses more efficient data structures (BDDs) for updating the nullary-predicate part of the abstract state. Further experiments are needed to precisely quantify the trade-off between the extra costs caused by the abstraction refinement loop of BLAST versus the global analysis of TVLA.

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 for this paper (search the web for the string "Lazy-Shape-Analysis-Supplementary-Material").

6 Related Work

Shape analysis based on three-valued logic is a framework that supports a family of abstractions [19,16,15], and from that standpoint, the predicate-abstraction component of our system simply contributes a set of nullary predicates to the shape-abstraction component. Thus our algorithm can be seen as generalizing (1) interpolation-based predicate discovery from nullary predicates to unary points-to and field predicates, and (2) lazy abstraction refinement from predicate abstractions to heap abstractions. We treat field predicates as core predicates, in contrast to [9], where abstractions of the data-structure contents are treated as instrumentation predicates. By relying on a given, fixed set of SCGs, we are still far from a completely automatic lazy implementation of shape analysis, which would require also the automatic discovery of more general instrumentation predicates. However, there are inherent limitations on what first-order theorem provers can deduce about three-valued abstractions that use transitive-closure predicates such as reachability [20]. Instead, one could use learning-based techniques from [17], which generate new instrumentation predicates that are not just boolean combinations of previously used predicates. These could be used in our system to dynamically add new SCGs.

There have also been proposals to encode shape analysis within predicate-abstraction frameworks [1,7]. So far they apply only to restricted settings, such as singly linked lists, or need user help for computing abstractions. Fischer et al. implemented in BLAST a combination of predicate abstraction with a parametric lattice-based data-flow analysis [8], but they did not consider any automatic refinement of the data-flow component. Gulavani and Rajamani proposed a CEGAR method for abstract interpretation and applied it to shape analysis [10], but their refinement is done globally, not lazily.

Acknowledgement. We thank Tom Reps for many valuable comments.

References

1. I. Balaban, A. Pnueli, and L.D. Zuck. Shape analysis by predicate abstraction. In *Proc. VMCAI*, LNCS 3385, pages 164–180. Springer, 2005.
2. T. Ball and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
3. D. Beyer, T.A. Henzinger, and G. Théoduloz. Lazy shape analysis. Technical Report MTC-REPORT-2005-006, EPFL, 2005.
4. S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.
5. D.R. Chase, M.N. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *Proc. PLDI*, pages 296–310. ACM, 1990.
6. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, LNCS 1855, pages 154–169. Springer, 2000.
7. D. Dams and K.S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Proc. VMCAI*, LNCS 2575, pages 310–324. Springer, 2003.
8. J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *Proc. ESEC/FSE*, pages 227–236. ACM, 2005.
9. D. Gopan, T.W. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Proc. POPL*, pages 338–350. ACM, 2005.
10. B.S. Gulavani and S.K. Rajamani. Counterexample driven refinement for abstract interpretation. In *Proc. TACAS*, LNCS 3920, pages 474–488. Springer, 2006.
11. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.
12. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
13. N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL*, pages 66–74, 1982.
14. R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
15. T. Lev-Ami, T.W. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Proc. ISSTA*, pages 26–38. ACM, 2000.
16. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proc. SAS*, LNCS 2280, pages 280–301. Springer, 2000.
17. A. Loginov, T.W. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *Proc. CAV*, LNCS 3576, pages 519–533. Springer, 2005.
18. K.L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, LNCS 2725, pages 1–13. Springer, 2003.
19. M. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 24(3):217–298, 2002.
20. G. Yorsh, T.W. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *ACM Trans. Comput. Log. (TOCL)*, to appear.