Abstraction for Shape Analysis with Fast and Precise Transformers

Tal Lev-Ami[1,⋆], Neil Immerman[2,⋆⋆], and Mooly Sagiv[1]

[1] School of Comp. Sci., Tel Aviv Univ.
{tla, msagiv}@post.tau.ac.il
[2] Dept. of Comp. Sci. UMass, Amherst
immerman@cs.umass.edu

Abstract. This paper addresses the problem of proving safety properties of imperative programs manipulating dynamically allocated data structures using destructive pointer updates. We present a new abstraction for linked data structures whose underlying graphs do not contain cycles. The abstraction is simple and allows us to decide reachability between dynamically allocated heap cells.

We present an efficient algorithm that computes the effect of low level heap mutations in the most precise way. The algorithm does not rely on the usage of a theorem prover. In particular, the worst case complexity of computing a single successor abstract state is $O(V \log V)$ where V is the number of program variables. The overall number of successor abstract states can be exponential in V. A prototype of the algorithm was implemented and is shown to be fast.

Our method also handles programs with "simple cycles" such as cyclic singly-linked lists, (cyclic) doubly-linked lists, and trees with parent pointers. Moreover, we allow programs which temporarily violate these restrictions as long as they are restored in loop boundaries.

1 Introduction

Automatically establishing safety properties of programs that permit dynamic storage allocation and low-level pointer manipulations is challenging. Dynamic allocation causes the state space to be infinite; moreover, a program is permitted to mutate a data structure by destructively updating pointer-valued fields of nodes.

It is well understood that reachability is crucial for reasoning about linked data structures. In this work we establish a simple abstraction method for reasoning about reachability that is provably efficient and precise. This provides both a practical analysis method and a theoretical contribution towards the understanding of how precise and efficient shape analysis can be.

1.1 Main Results

We present a method to conservatively verify reachability properties via abstract interpretation [4]. Specifically, we present a new lightweight method for shape analysis (e.g.,

[⋆] Supported by an Adams Fellowship through the Israel Academy of Sciences and Humanities.
[⋆⋆] Supported by NSF grant CCF-0514621.

T. Ball and R.B. Jones (Eds.): CAV 2006, LNCS 4144, pp. 547–561, 2006.

see [10,21]) that applies to programs on "regular tree-like" data structures. The method is sound, i.e., whenever it reports that a safety property holds, it indeed holds. Furthermore, we compute **the best abstract transformer** [4] for atomic Java-like statements. A prototype of the algorithm was implemented and is shown to be fast. The system can be seen as a specialization of TVLA [16] to a set of data-structures and a set of properties.

In the rest of the section, we elaborate on the key contributions. Sect. 8 includes more detailed comparison to related work.

New Abstraction of Heap Shape. In Sect. 3, we present our simple abstraction for heaps based on contracting segments of the heap into a single summary-node.

In contrast to existing methods, our abstraction admits the precise and efficient recovery of reachability information concerning the modeled concrete states. For example, every path in the abstraction between "important" nodes is a must-path, i.e., it must exist between the corresponding nodes in each modeled concrete state. Thus, reasoning about reachability between important nodes can be performed efficiently via simple graph traversal.

We show that the abstraction of graphs with no undirected cycles yields a linear number of nodes. Therefore, the size of the abstract state space is bounded for such programs, allowing effective state space exploration. Moreover, this also holds for simple cycles such as cyclic singly-linked lists, (cyclic) doubly-linked lists and trees with parent pointers. Furthermore, it is possible to apply our abstraction only in loop boundaries and thus allowing programs to temporarily violate the data-structure invariants. Full proofs for the theorems in the paper can be found in [15].

Efficient Best Transformers. In Sect. 4, we present an efficient $O(NS * V * \log V)$ algorithm for computing the best abstract transformers for Java-like atomic program statements including destructive pointer manipulation. V is the number of program variables. NS is the number successor abstract states (can be exponential in the number of program variables).

Most existing methods for shape analysis including TVLA do not implement the best transformers and may require exponential time to produce a single abstract state. Also, in contrast to existing methods for generating the best abstract transformers (e.g. [7,22,2]), our method does not employ a theorem prover. Precise reachability information is maintained using our abstraction.

Efficient algorithms for computing the best transformers for predicate abstraction in singly-linked lists were developed in [18]. This paper can be considered as a continuation of [18] that handles more complex data-structures.

Information Extraction. It is important to extract information from an abstract state about the concrete states that it models. For example, we sometimes need to verify disjointness of data structures. For safety properties we check that user-specified assertions hold in every execution leading to a given program point.

In Sect. 5, we provide a conservative and efficient method that extracts such information by evaluating a first-order formula with transitive closure on a given abstract state. Our method is more precise than standard Kleene evaluation (e.g., [21]), although less precise than supervaluational semantics [3,19]. We show that our method is exact for "atomic" reachability properties between important nodes. Our limited experiments indicate that one of our evaluation methods is precise enough in practice.

2 Preliminaries

We call an allocated object on the heap a **heap node**. Shape analysis tracks reference program variables and reference fields, i.e., to which heap node each reference variable points to and for each heap node where each of its reference fields point to. In this paper we assume a fixed set of (reference) program variables denoted by $PVar$ and a fixed set of reference fields denoted by $PRef$.

A **state** (shape graph [10]) is a triple $C \overset{\text{def}}{=} (U^C, env^C, ref^C)$. The universe, U^C, is the set of allocated heap nodes; the environment, $env^C \subseteq PVar \times U^C$, is a partial function from program variables to the heap nodes that they point to; and $ref^C : PRef \to \mathcal{P}(U^C \times U^C)$ is a function from each field name f to a relation which pairs each node with the node its f field points to. Since these relations induce a graph on the heap nodes, we will use the term f**-edge** for a pair of nodes in the relation $ref^C(f)$ and call f its **edge type**. In languages such as Java where the program cannot use the memory address of an object directly, the specific names of the nodes in U^C are immaterial. Thus, we define equality between states as isomorphism between them.

2.1 Notations

Fig. 1 lists some notation used throughout. We shorten $E\{\{x\}\}$ to $E\{x\}$.

Symbol	Definition	Meaning
E^*	Reflexive Transitive Closure of E	
$succ(X, E)$	$\{(n, n') \in E \mid n \in X\}$	Restriction of first component
$pred(X, E)$	$\{(n, n') \in E \mid n' \in X\}$	Restriction of second component
$E_1 \circ E_2$	$\{(n, n'') \mid (n, n') \in E_2, (n', n'') \in E_1\}$	Relation composition
$E\{X\}$	$\{n' \mid (n, n') \in succ(X, E)\}$	Relation image
$up_{b \to a}$	$\lambda n.$ if $(n = b)$ then a else n	Updating b to be a
fld^C	$\bigcup_{f \in PRef} ref^C(f)$	Edges of C
$disj(v_1, v_2, v_3)$	$v_1 \neq v_2 \wedge v_1 \neq v_3 \wedge v_2 \neq v_3$	The variables are disjoint

Fig. 1. Notations used in the paper

We define $var(S) \overset{\text{def}}{=} env^S\{PVar\}$ to be the set of nodes in S pointed to by program variables and $shared(S)$ to be that set of nodes in S that are pointed to by two or more different heap nodes (ignoring self-loops). We say such a node is **heap-shared**. Formally, $shared(S) \overset{\text{def}}{=} \{v \mid (w_1, v) \in fld^S, (w_2, v) \in fld^S, disj(w_1, w_2, v)\}$

3 Abstraction

A state, C, is **concrete** if none of its edges are self loops and if each $ref^C(f)$ is a partial function. The main idea of the abstraction is to keep a set of distinct nodes which are not abstracted and abstract the rest of the graph in such a way that keeps all reachability information for these nodes explicit. The set of distinct nodes we use are those nodes that are either pointed to by variables or heap shared, i.e., $distinct(S) \overset{\text{def}}{=} var(S) \cup shared(S)$.

We **contract** an edge (a, b) by replacing each occurrence of b by a, $contract(S, a, b) \overset{\text{def}}{=} (U^S - \{b\}, env^S, \lambda f.\{(up_{b\to a}(n_1), up_{b\to a}(n_2) \mid (n_1, n_2) \in ref^S(f)\})$ (note that env^S is not updated because we never contract a node pointed to by a variable). We now define a method $\mathcal{B}(S, D)$ that given a state and a set of nodes D s.t. $distinct(S) \subseteq D \subseteq U^S$, returns the abstract state generated by repeatedly applying contraction on all edges that are not incident to nodes in D until the unique fixpoint is reached. An equivalent way to define $\mathcal{B}(S, D)$ is by collapsing every maximal connected subgraph T_n of S that does not contain nodes in D (the subgraph is a rooted tree) to a single node n (its root). The edge types of the self-loops of n are exactly the types of edges within T_n.

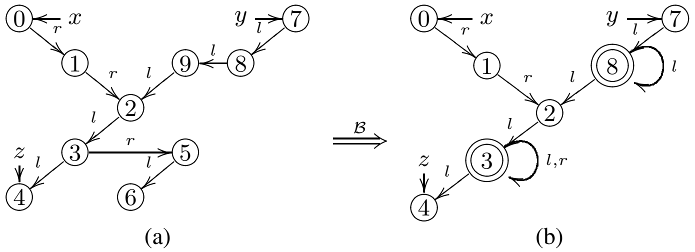

Fig. 2. (a) A concrete state C_1, (b) $S_1 = \mathcal{B}(C, distinct(C_1))$

We call the function, M, that maps each node to the node it was collapsed into by $\mathcal{B}$ the **embedding function** (after [21]). When multiple nodes have been embedded into a single node n (i.e., $|M^{-1}(n)| > 1$) we call n a **summary node**. Fig. 2 gives an example of a concrete state C_1 and the result of $\mathcal{B}(C_1, distinct(C_1))$. We mark summary nodes with a double-circle for emphasis.

The abstraction relation, $\beta \overset{\text{def}}{=} \{\, (S, \mathcal{B}(S, distinct(S))) \mid S \text{ a state}\}$, maps each state, S, to a state in which every edge not incident to a distinct node has been contracted.

3.1 Data Structures

We limit the class of data structures handled to graphs with no undirected cycles (i.e., when we remove the direction of the edges we get an undirected forest) and no garbage (i.e. all nodes are reachable from program variables). We call such states **admissible states**. This class includes linked lists, trees, and trees with limited amount of sharing (i.e., each pair of nodes has at most one simple path between them and each pair of variables meets at most once). Extensions to support cyclic linked lists, doubly linked lists and trees with parent pointers are described in Sect. 6.

We use a standard relational abstract domain with set-union as join (in Sect. 6 we define a more concise partial-join operator). The concretization relation is defined as

$\gamma \overset{\text{def}}{=} \{(S, C) | (C, S) \in \beta$ and C is an admissible concrete state$\}$. We say that an abstract state, S, is **feasible** if $\gamma\{S\} \neq \emptyset$, i.e. S models some admissible concrete state.

3.2 Properties of the Abstraction

We start with some important definitions:

- We say that $(n_1, n_2) \in ref^S(f)$ is an f **must edge** when
  $\forall C \in \gamma\{S\}, n_1' \in M^{-1}(n_1), n_2' \in M^{-1}(n_2) \,.\, (n_1', n_2') \in ref^C(f)$
- We say that $(n_1, n_2) \in ref^S(f)$ is an f **may edge** when
  $\forall C \in \gamma\{S\} \,.\, \exists n_1' \in M^{-1}(n_1), n_2' \in M^{-1}(n_2) \,.\, (n_1', n_2') \in ref^C(f)$
- We say that $(n_1, n_2) \in ref^S(f)$ is an f **unique may edge** when
  $\forall C \in \gamma\{S\} \,.\, \exists!\, n_1' \in M^{-1}(n_1), n_2' \in M^{-1}(n_2) \,.\, (n_1', n_2') \in ref^C(f)$
- We say there is a **must path** between n_1 and n_2 when
  $\forall C \in \gamma\{S\}, n_1' \in M^{-1}(n_1), n_2' \in M^{-1}(n_2) \,.\, (n_1', n_2') \in (fld^C)^*$
- We say there is a **may path** between n_1 and n_2 when
  $\forall C \in \gamma\{S\} \,.\, \exists n_1' \in M^{-1}(n_1), n_2' \in M^{-1}(n_2) \,.\, (n_1', n_2') \in (fld^C)^*$

The abstract state S_2 in Fig. 3 models all singly-linked lists of length 4 or more s.t. x points to their head and y to their tail. Note that there are cases in which there is a must path between two nodes (e.g. from 0 to 2) although the path in the abstract state contains may edges (the edges from 0 to 1 and from 1 to 2).

Thm. 1 summarizes some important properties of our abstraction.

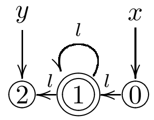

Fig. 3. An abstract state S_2

Theorem 1. *For every feasible abstract state S the following hold:*
1. Every f edge in S is an f may edge.
2. Every non self-loop f edge is an f unique may edge.
3. Every f edge between non-summary nodes is an f must edge.
4. A node in S is a summary node iff it has self-loops.
5. For every summary node n the subgraph induced by $M^{-1}(n)$ is a tree and has a unique incoming edge which leads to its root.
6. Let $n_1 \neq n_2$ where n_1 has no self-loops or a single self-loop of the same type as its outgoing edge. A path from n_1 to n_2 is a must path.

Proof: *(sketch)*

1. Immediate from definition of contraction.

2. Analysis of possible contractions reveals that the only case in which two edges are merged by a contraction is if an undirected cycle appeared in the original state.

3. Immediate from 4 and definition of contraction.

4. Contraction always creates a self-loop. Self-loops are preserved by contraction and contraction is the only way to create self-loops.

5. Let T_n be the subgraph induced by $M^{-1}(n)$. Since contraction is done on edges, the nodes in T_n are weakly connected. Shared nodes are never contracted, thus there is no sharing in T_n. Since the original state had no garbage any cycle either has a variable pointing to it, or has a shared node. In any case, an entire cycle cannot be contracted to the same summary node. Thus, T_n is a tree. Furthermore, to avoid sharing and garbage, the one and only incoming edge must be to its root.

6. By 5 and 1, every summary node represents a tree and every edge is a may-edge. Thus, paths between non-summary nodes are must paths. Since a summary node is a tree, all the nodes in it are reachable from the root and so if the target node is a summary node, the path is still a must path. If the source node has a single self-loop it is a singly-linked list. The only outgoing edge from a singly-linked list of the same type as the self-loop is from its last node, thus reachable from all nodes. □

The last property is of particular importance since it means that the reachability information in the abstract state is explicit. This property is not standard in shape analysis abstractions (e.g., in TVLA it in not always the case). The reason for the limitation on n_1 is that if n_1 has 2 or more self-loops it embeds a tree, thus n_2 is not reachable from some nodes embedded to n_1 (e.g. in Fig. 2 the path in S_1 from node 3 to node 4 is not a must path, since for example in C_1 there is no path from node 5 to node 4).

Lem. 1 defines when an abstract state S is feasible and Lem. 2 bounds its size. Note that the set of admissible concrete states is exactly the set of feasible abstract states with no self-loops.

Lemma 1. (Feasibility) *Abstract state S is feasible iff the following hold:*
1. There are no edges between two different non-distinct nodes.
2. Distinct nodes are never summary nodes.
3. A node that has two outgoing f edges has a self-loop of a different edge type.
4. Deleting all self-loops from S makes it admissible.

Proof: *(sketch)*

(Only If) 1. An edge between two different non-distinct nodes can be contracted, which contradicts that S is in the image of β.

The rest of the properties hold in concrete admissible states and are preserved by contraction.
2. Immediate from definition of contraction.
3. A counterexample would be a node with zero or one self-loops and two outgoing edges of the same type. Since in the original concrete state each edge is a partial function, a node without self-loops cannot have two outgoing edges of the same type. A node with a single self-loop is a linked list, thus the only outgoing edge from it can be from its tail, thus a single edge.
4. It is easy to see that contractions do not introduce garbage or undirected cycles (except for self-loops).

(If) It can be shown that a state that satisfies these properties can always be expanded to a concrete state of finite size. □

Lemma 2. (MaxSize) *For every feasible abstract state S we have $|U^S| \leq MaxSize$, where $MaxSize \overset{\text{def}}{=} (|PRef| + 1) * (2 * |PVar| - 1)$*

Proof: Let C be an admissible concrete state, D a set s.t. $var(C) \subseteq D \subseteq U^C$, and $S = \mathcal{B}(C, D)$. S has the property that every node is either in D or has a parent in D. Thus, the number of nodes in $|U^S| \leq |D| * (|PRef| + 1)$. Since C has no garbage and no undirected cycles, $|distinct(C)| \leq |PVar| * 2 - 1$. Thus, if $(C, S) \in \beta$ then $|U^S| \leq MaxSize$. □

4 Best Transformers

Concrete Semantics. Fig. 4 defines the concrete semantics for simple atomic statements in Java-like programs. Most preconditions were added to simplify the presentation. In practice we use temporaries to translate each program statement to a sequence of operations while maintaining the preconditions. Some preconditions such as no null-dereference cannot be removed by a sequence of operations. The analysis detects violations of these preconditions and gives a warning.

The gc operation performs garbage collection by removing all nodes not reachable from any variable. Garbage collection can by executed either after every $x = null$ operation, periodically, or we can run garbage detection instead of garbage collection to detect memory leaks. The semantics of the other operations are straightforward formalizations of standard Java-like operational semantics.

Operation	Precondition	Semantics
gc		let $R = fld^*(var(S))$ in $(R, env, \lambda f.succ(R, ref(f)))$
$x = null$		$(U, succ(PVar - \{x\}, env), ref)$
$x = y$	$env\{x\} = \emptyset$	$(U, env \cup \{(x, n) \mid (y, n) \in env\}, ref)$
$x.f = null$	$env\{x\} \neq \emptyset \wedge$ $ref(f) \circ env\{x\} \subseteq var(S)$	$(U, env,$ $ref[f := succ(U - env\{x\}, ref(f))])$
$x.f = y$	$env\{x\} \neq \emptyset \wedge$ $ref(f) \circ env\{x\} = \emptyset$	$(U, env, ref[f := ref(f) \cup \{(n_x, n_y) \mid$ $(x, n_x) \in env, (y, n_y) \in env\}])$
$x = y.f$	$env\{x\} = \emptyset \wedge$ $env\{y\} \neq \emptyset$	$(U, env \cup \{(x, n) \mid (y, n) \in ref(f) \circ env\},$ $ref)$
$x = malloc$	$env\{x\} = \emptyset \wedge n_{malloc} \notin U$	$(U \cup \{n_{malloc}\}, env \cup \{(x, n_{malloc})\}, ref)$
$x == null$		$env\{x\} = \emptyset$
$x == y$		$env\{x\} = env\{y\}$

Fig. 4. The operations supported and their concrete semantics

4.1 Abstract Transformers

We now show how to compute the best abstract transformers (see [4]) for the our abstraction and concrete semantics defined in Fig. 4. The best transformer of an operation st is defined as $st^{best} \stackrel{\text{def}}{=} \beta \circ st \circ \gamma$ (i.e, for each concrete state in $\gamma\{S\}$ apply the concrete semantics and abstract). This definition is not constructive since the number of states in $\gamma\{S\}$ is unbounded and potentially infinite. The main idea is to define a relation $focus[st]$ whose image is a bounded set of states and if $(S, S') \in focus[st]$ there is a representative state $C \in \gamma\{S\}$ s.t. $\beta\{st(S')\} = \beta\{st(C)\}$ and vice versa. Thus, we define the abstract transformer to be $st^\sharp \stackrel{\text{def}}{=} \beta \circ st \circ focus[st]$. Note that the transformer defined in the concrete semantics can be applied to abstract states as well.

The focus operation is similar to the one defined in [21], i.e., it is a partial concretization intended to restore enough information to compute the transformer precisely. Let $D(st, C) \stackrel{\text{def}}{=} distinct(C) \cup distinct(st(C))$. We define focus to be:

Definition 1. $focus[st] \overset{\text{def}}{=} \{(S, \mathcal{B}(C, D(st, C))) \mid C \in \gamma\{S\}\}$

Focus takes all the states in $\gamma\{S\}$ and keeps both the distinct nodes of the state and the nodes that will become distinct after the statement is executed. In Sect. 4.2 we define an algorithm that computes the image of focus.

Lem. 3 gives some important properties for the interaction of β and st. Note that the existence of commutative diagrams is not true in general shape abstraction. Thm. 2 uses Lem. 3 to prove that $st^\sharp$ is the best abstract transformer.

Lemma 3. *For every* $(S, C) \in \gamma$*, let* $D = D(st, C)$ *and* $S' = \mathcal{B}(C, D)$*. Then:*
Idempotence. $\beta\{S'\} = \beta\{S\}$*,*
Commutative Diagrams. $\mathcal{B}(st(C), D) = \mathcal{B}(st(S'), D)$*, and*
Equivalence under β**.** $(\beta \circ st)\{C\} = (\beta \circ st)\{S'\}$

Proof: *(sketch)*
Idempotence. It can be shown that contraction induces a confluent derivation relation commutative in the choice of D. Since $\mathcal{B}$ can be seen as the fixed-point of that relation, the statement follows.
Commutative Diagrams. This can be verified by checking the algebraic operations defining the transformer, for each operation in Fig. 4.
Equivalence under β**.** By commutative diagrams we have $\mathcal{B}(st(C), D)$=$\mathcal{B}(st(S'), D)$. By Idempotence we have $\beta\{\mathcal{B}(st(C), D)\} = \beta\{st(C)\}$ and $\beta\{\mathcal{B}(st(S'), D)\} = \beta\{st(S')\}$. Thus, $\beta\{st(C)\} = \beta\{st(S')\}$. □

Theorem 2. $st^\sharp$ *is the best abstract transformer, i.e.,* $st^\sharp = st^{best}$.

Proof: Let $(S, S^\sharp) \in st^\sharp$. There is S' s.t. $(S, S') \in focus[st]$ and $(S', S^\sharp) \in \beta \circ st$. By Def. 1 there is a concrete and admissible state C s.t. $(S, C) \in \gamma$ and $S' = \mathcal{B}(C, D(st, C))$, and by Lem. 3 $(C, S^\sharp) \in (\beta \circ st)$ thus $(S, S^\sharp) \in st^{best}$.

Conversely, let $(S, S^\sharp) \in st^{best}$. There is C s.t. $(S, C) \in \gamma$ and $(C, S^\sharp) \in (\beta \circ st)$. Let $S' = \mathcal{B}(C, D(st, C))$. By Def. 1 we have $(S, S') \in focus[st]$, and by Lem. 3 $(S', S^\sharp) \in \beta \circ st$ thus $(S, S^\sharp) \in st^\sharp$. □

4.2 Algorithms

In order to compute the best abstract transformer, $st^\sharp$, we must give efficient algorithms for state equality, focus, and β. The total complexity of computing the abstract transformer is $O(NS * V * \log V)$ where NS is the number of successor abstract states (which may be exponential in the number of program variables).

Focus. In Sect. 4.1, we defined focus non-constructively. We now present an algorithm, $Focus(S, st)$, that computes $focus[st]\{S\}$. The first observation is that for all statements, st, except $x = y.f$, $focus[st]$ is the identity relation. This is clear for $x = malloc$, and true for the rest because $distinct(st(S)) \subseteq distinct(S)$,

For $st \overset{\text{def}}{=} x = y.f$, $Focus(S, st)$ enumerates on all states that can be contracted to S by a minimal number of contractions and still have $distinct(st(S)) \subseteq distinct(S)$ as non-summary nodes. Let n_f be the node pointed to by $y.f$ in S. If it is a non-summary node $Focus(S, st) = \{S\}$. Otherwise, let G be the self-loops of n_f in S.

Let $(S, S') \in focus[x = y.f]$, S' can be contracted into S by at most one contraction for each edge type in G. Let N'_f be the subgraph of S' that was contracted into n_f. Since all edges are may-edges, the edges within N'_f are exactly the self-loops of n_f. Furthermore, since all the edges between different nodes are unique may-edges, the edges between N'_f and the rest of the graph are exactly the edges between n_f and the rest of the graph. Finally, since S' is the result of $\mathcal{B}$ on an admissible concrete state the property that a node that has two outgoing g edges has a self-loop of different reference field, is maintained. This gives us an enumeration algorithm to compute $Focus(S, st)$. Lem. 4 summarizes the properties of $Focus(S, st)$.

Lemma 4. $focus[x = y.f]\{S\} = Focus(S, x = y.f)$

Beta. To compute the image of β we perform two tasks, 1) check that the state is admissible and 2) return a state in which all the possible contractions have been made.

Admissibility. Since an admissible state is one without garbage and with no undirected cycles, the check is done by DFS from all nodes pointed to by variables to make sure that there is no garbage. To compute undirected connectivity, we maintain a Union-Find data structure during the DFS, thus detecting undirected cycles. We start with singleton groups for each node and for every edge we encounter we union the groups the two incident nodes belong to. Thus the sets maintain weak reachability. If we find the two incident nodes already belong to the same group we found an undirected cycle and we abort. The complexity for this check is $O(n\alpha(n))$, where n is the size of the input state and α is the inverse Ackerman function.

To compute $\beta\{S\}$ we observe that the edges contracted are exactly the edges between non-distinct nodes. Thus, the algorithm performs two DFS traversals. The first computes $distinct(S)$ by marking nodes that are either pointed to by variables or have an in-degree greater than one (note that self-loops do not contribute to the in-degree). The second traversal simply contracts every non self-loop edge s.t., both its incident nodes are not distinct. The complexity of this algorithm is $O(n)$.

State Equality. We defined state equality as isomorphism between the states. We give an algorithm that computes canonical names for each state. The canonical names of two states are identical iff the two states are isomorphic.

Canonical names are given to nodes by traversing the graph in DFS from program variables (in fixed order) traversing the reference fields in fixed order as well. The name of a node n is composed of the names of the variables pointing to n, n's self loops and for each of n's parents, the parent name and the type of the edge leading from the parent to n. To ensure the traversal order is unique, we only leave a node to its children after all its parents have been visited. Hash-cons is used to store the canonical names, allowing for O(1) amortized time equality checks. The name of a state is the hash-cons of its set of nodes ordered by some fixed order (e.g. memory address of the hash-cons). Thus, the total complexity of the algorithm is $O(V \log V)$.

5 Evaluation

We use a subset of first-order logic with transitive closure as a query logic to extract information from states. Let $[\![\varphi]\!]^S$ denote the boolean value of formula φ in state S.

Definition 2. (Sound) *An evaluation function of a formula is* ***sound*** *iff for every feasible abstract state S, $\neg[\![\varphi]\!]^S \Rightarrow \forall C \in \gamma\{S\} \,.\, \neg[\![\varphi]\!]^C$*

(Complete) *An evaluation function of a formula is* ***complete*** *iff for every feasible abstract state S, $\neg[\![\varphi]\!]^S \iff \forall C \in \gamma\{S\} \,.\, \neg[\![\varphi]\!]^C$*

To compute `assert`(φ, S), i.e., to verify that all the states in $\gamma\{S\}$ satisfy φ, we will apply a sound evaluation function on $\neg\varphi$ and verify that the result is false.

5.1 Query Logic

The query logic is first order logic in Negation Normal Form (NNF) over the following vocabulary:

- For every $x \in PVar$ a unary predicate symbol; $x(n)$ iff x points to n
- For every $f \in PRef$ a binary predicate symbol; $f(n_1, n_2)$ iff the f field of the n_1 points to the n_2
- Binary predicate symbol TC; $\mathrm{TC}(n_1, n_2)$ iff there is any non-empty path from n_1 to n_2
- Equality; $n_1 = n_2$ iff n_1 and n_2 are the same heap node

Examples:

$$\forall v \,.\, \exists w . x(w) \wedge (v = w \vee \mathrm{TC}(w, v)) \tag{1}$$

$$\forall v, w \,.\, \neg y(w) \vee \neg left(v, w) \tag{2}$$

Formula (1) states that all the nodes in the heap are either pointed to by x or reachable from the node pointed to by x. Formula (2) states that the any node pointed to by y has no incoming $left$ edge.

We will restrict our attention to closed formulas (no free variables). We say that a formula is **guarded** if every quantifier is of the form $(\forall v \,.\, x(v) \Rightarrow \psi)$ or $(\exists v \,.\, x(v) \wedge \psi)$ where x is some program variable.

To evaluate formula φ in state S we translate S to a standard logical structure $\widehat{S}$ and φ to a FO formula, $TR(\varphi)$, in the vocabulary of $\widehat{S}$. Let $[\![\varphi]\!]^S \overset{\text{def}}{=} [\![TR(\varphi)]\!]^{\widehat{S}}$ where the right hand side is standard FO Tarskian semantics. Thm. 3 ensures the soundness of the evaluation and guarantees completeness for the guarded fragment of the query logic.

Theorem 3. *For every formula φ, $\lambda S.[\![TR(\varphi)]\!]^{\widehat{S}}$ is a sound evaluation function. If φ is guarded, it is also a complete evaluation function.*

5.2 Translation

The universe of $\widehat{S}$ is the universe of S. The vocabulary and its interpretation are given in Fig. 5(a). The translation defines for each edge f two predicates, $f^{\forall}$ and $f^{\exists}$. If $f^{\forall}(n_1, n_2)$ then there is an f must edge from n_1 to n_2. If $f^{\exists}(n_1, n_2)$ then there is an f may edge from n_1 to n_2. Similarly we use $\mathrm{TC}^{\forall}(n_1, n_2)$ to define a must path from n_1 to n_2, and $\mathrm{TC}^{\exists}(n_1, n_2)$ to define a may path from n_1 to n_2. The translation is a formalization of Thm. 1. Fig. 5(b) gives the translation of S_2 defined in Fig. 3. The translation rules for the literals in the query formula are given in Fig. 5(c).

Vocabulary	Interpretation
$x(n)$	$(x, n) \in env^S$
$f^{\exists}(n_1, n_2)$	$(n_1, n_2) \in ref^S(f)$
$f^{\forall}(n_1, n_2)$	$f^{\exists}(n_1, n_2) \wedge \neg sm(n_1) \wedge \neg sm(n_2)$
$\mathrm{TC}^{\exists}(n_1, n_2)$	A (possibly empty) directed path from n_1 to n_2
$\mathrm{TC}^{\forall}(n_1, n_2)$	$\mathrm{TC}^{\exists}(n_1, n_2)$, $n_1 \neq n_2$ and the path satisfies case 6 of Thm. 1
$sm(n)$	$\bigvee_{f \in PRef}(n, n) \in ref^S(f)$

(a)

Predicate	Tuples
x	$\langle 0 \rangle$
y	$\langle 2 \rangle$
$left^{\exists}$	$\langle 0, 1 \rangle, \langle 1, 1 \rangle, \langle 1, 2 \rangle$
$left^{\forall}$	
$\mathrm{TC}^{\exists}$	$\langle 0, 1 \rangle, \langle 0, 2 \rangle, \langle 1, 1 \rangle, \langle 1, 2 \rangle$
$\mathrm{TC}^{\forall}$	$\langle 0, 1 \rangle, \langle 0, 2 \rangle, \langle 1, 2 \rangle$
sm	$\langle 1 \rangle$

(b)

φ	$TR(\varphi)$	φ	$TR(\varphi)$
$x(v)$	$x(v)$	$\mathrm{TC}(v_1, v_2)$	$\mathrm{TC}^{\exists}(v_1, v_2)$
$\neg x(v)$	$\neg x(v)$	$\neg \mathrm{TC}(v_1, v_2)$	$\neg \mathrm{TC}^{\forall}(v_1, v_2)$
$f(v_1, v_2)$	$f^{\exists}(v_1, v_2)$	$n_1 = n_2$	$n_1 = n_2$
$\neg f(v_1, v_2)$	$\neg f^{\forall}(v_1, v_2)$	$\neg n_1 = n_2$	$\neg n_1 = n_2 \vee sm(n_1)$

(c)

Fig. 5. (a) Translation of an abstract state to a logical structure. (b) $\widehat{S_2}$ - the translation of S_2 from Fig. 3 (c) Rules for translating a query formula to the vocabulary of $\widehat{S}$.

Theorem 3 Proof: *(sketch)*
The evaluation of $TR(\varphi)$ on $\widehat{S}$ simulates the evaluation of a φ on any concrete state C s.t. $(S, C) \in \gamma$. Assume an assignment $v_i \mapsto n_i$ satisfies a literal $L(v_1, ..., v_k)$ in S', we shall see that $v_i \mapsto M(n_i)$ satisfies $TR(L)(v_1, ..., v_k)$. Most cases are immediate from the definition of $\widehat{S}$ and the properties of the abstraction (Sect. 3.2). The only case requiring further explanation is $L \equiv \neg v_1 = v_2$. Here we may chose $n_1 \neq n_2$ s.t. $M(n_1) = M(n_2)$, but in this case $sm(M(n_1))$ thus $TR(L)(v_1, v_2)$ still evaluates to true. Since an NNF formula has no negation outside of literals this is enough for soundness. □

Examples: The translation of (1) is $\forall v.\exists w.x(w) \wedge (v = w \vee \mathrm{TC}^{\exists}(w, v))$ which evaluates to true in $\widehat{S_2}$ as expected. The translation of (2) is $\forall v, w.\neg y(w) \vee \neg left^{\forall}(v, w)$ unfortunately this formula also evaluates to true. In some cases, including this one, we can overcome this imprecision by an improved formula translation $TR'(\varphi)$, as described in [15].

6 Extensions

6.1 Loop Boundaries

Some programs temporarily violate the data structure invariants (including admissibility) and restore all within the boundary of a single loop iteration. We can handle such programs with the same level of precision by only performing β on loop boundaries.

6.2 Partial Join

Partial Join [17] replaces union as the join operator of the abstract domain with an operator that merges matching states. We build a variant of the partial join operator by ignoring the self-loops when giving canonical names to states. Matching states are merged by performing union on the self-loops on nodes with the same canonical names. The concretization function is modified to consider that some of the self-loops may not represent concrete edges.

The focus operation needs to be updated according to the changes in the concretization function. There are two changes in the algorithm: 1) There is no need to enumerate the self-loops in the subgraph contracted to the summary node. 2) The case in which the summary node represents a single node needs to be considered.

The experimental results (Sect. 7) show that Partial Join is important for performance, while maintaining precision.

6.3 Cycles

The abstract domain can be extended to support cycles in the following limited way. A directed cycle is admissible if there is a path from a variable that contains the entire cycle and all the outgoing edges from all the nodes of this path are of the same edge type (i.e. the cycle is a part of a singly-linked list). A state is admissible if all its undirected cycles are actually admissible directed cycles. All the properties of the abstraction such as the bounded abstract state size remain true for this extended class.

Focus and β can be easily modified to support these cycles since an entire cycle can never be contracted (since there has to be a node on each cycle that is either pointed to by a variable or heap-shared). The subtleties come from two sources. One is the fact the a self loop can now represent a concrete self-loop and not a summary node. This can be easily solved by adding an extra bit per node indicating whether it is a summary node or not and maintaining it in all the operations.

The second subtlety is in computation of canonical names, since without breaking the cycles we may never be able to give a name to a node before traversing its children. The solution is to mark the back-edges during the first DFS and ignore them in the second DFS. At the end, we add their names to their incoming nodes.

6.4 Parent Pointers

The abstract domain can be extended to allow parent pointers (i.e., doubly linked lists and trees with parent pointers) in the following limited way. Each node can use only a single field as a parent pointer (specified by the user). Parent pointers are not considered for contraction, heap-sharing or garbage (thus every node has to be reachable using non parent-pointer fields). This means that exactly the same nodes will be contracted whether parent pointers exist or not. Either all the nodes contracted to a summary have the same parent pointer (in this case we say that the summary node has that parent pointer) or none of them have it. If two nodes are contracted, all the parent pointers incoming or outgoing from these nodes have to be the inverse of "real" reference fields and the two nodes and the edge between them have to agree on the parent pointer (either none have parent pointers, or all of them have the same parent pointer).

These limitations still allow us to handle doubly-linked lists and trees with parent pointers as long as every node is reachable using "real" reference fields (i.e. there is a pointer from the head of the doubly linked list or from the root of the tree). Specifically we can handle all the doubly-linked list examples of [21].

To support this extension we make the following changes:

Focus. The only problem in the current focus is the fact that we can now traverse a parent pointer into a summary node and, in this case, it does not necessarily lead to the root of the sub-graph contracted to the summary node. The parent pointers within the sub-graph are easy to handle since they are either the inverse of all the reference fields in the sub-graph or none of them.

Beta. Since the contractions ignore the parent pointers we only need to make sure that the state is admissible. We update the current admissibility check to consider the parent pointer limitation described above.

Updating the canonical names algorithm is simple as well.

7 Implementation

We have implemented the abstract transformer detailed above including the extensions of Sect. 6. Focus was implemented only for linked lists and binary trees (i.e., up to two self-loops). The implementation is written in Java and is integrated with the Soot Java Optimization Framework [20] as a front end. The empirical results of running our analysis on some examples are given in Fig. 6. In all cases the analysis also proved absence of memory leaks, acyclicity (where applicable) and absence of null-dereferences. N/A states that the information for the example is not available for that tool and O/S means that it is out of scope for the tool. Max states is the maximum number of states in each program point. The columns marked with ”[R]” use the relational join as described in Sect. 4. The columns marked with ”[P]” use the partial join extension described in Sect. 6. The TVLA times given for tree manipulating algorithms use partial join as well. The tests were made on an Intel Pentium M, 1.6 GHz with 1.00 GB of RAM.

Programs	Time[R]	Max states[R]	Time[P]	Max states[P]	[2]	TVLA	[18]
deleteSortedTree	2359.70	192355	3.22	520	O/S	47.48	O/S
insertSortedTree	20.85	9365	0.55	264	O/S	1.8	O/S
lindstromScan	1459.63	79673	8.36	1337	O/S	65.86	O/S
insertRedBlack	> 24 hours		38.15	4853	O/S	N/A	O/S
reverse	0.05	15	0.11	8	0.1	0.531	5
reverseCycle	0.24	159	0.26	62	0.1	N/A	2
merge	0.20	96	0.15	36	17.8	4.006	15
delete	0.02	20	0.01	12	0.9	1.242	7
bubbleSort	0.03	36	0.03	21	N/A	11.887	N/A
bubbleSort2	0.08	76	0.08	33	11.4	N/A	N/A
insertSort	0.06	100	0.05	48	N/A	20.219	N/A

Fig. 6. The empirical results from running the abstract transformer implementation

The programs are explained in [15]. The “bubbleSort” and “bubbleSort2” are two variants of an in-place bubble sort for linked lists analyzed by TVLA and [2] respectively.

We can see that our analysis is indeed fast and in some cases up to 100 times faster than the other analyses depicted. We should point out that most examples are small, thus the differences in running times can be partially attributed to engineering issues. Checking the properties detailed above for these examples is done automatically by the system. To check other properties we need a way to extract information from the abstract states. This is done by formula evaluation and is detailed in Sect. 5.

8 Related Work

Shape and heap analysis is a subject of active research with many interesting algorithms including [10,21,13]. The TVLA system generalizes these algorithms and can be utilized to implement our algorithm. Indeed, in this paper we followed the line of research similar to the one in [8,13,12,18] of developing a specialized shape analysis for commonly used data structures.

We are very pleased with the ability of our method to compute the best transformers in an efficient way. In contrast, TVLA can spend a lot of time in order to determine if an abstract state is feasible. Indeed it can spend an exponential time even when there are no resultant abstract states. The abstraction in this paper is tailored for an interesting set of properties. A mechanism to support other properties (such as TVLA's Instrumentation Predicates) remains an interesting open problem.

Connection analysis [6] keeps reachability information between program variables. Our work is more precise as it can perform strong updates for heap manipulation. Grammar based abstraction [13] uses a restricted grammar to annotate summary nodes with their possible shapes. The abstractions are incomparable since the grammar based abstraction can express invariants (such as binomial heap) that cannot be expressed in our abstraction. On the other hand, the grammar based abstraction can deal with only a limited amount of sharing. For example, it cannot represent a tree with parent pointers and a pointer arbitrarily deep into the tree.

The shape analysis of [5] is very similar to [18] both in the properties of the abstraction and in the programs handled.

Decision Procedures for Linked Data Structures. An orthogonal line of research is the development of decision procedures and theorem provers which support transitive closure [1,9,14,2]. Such techniques can be utilized with arbitrary abstractions.

In this paper, we developed direct methods for a specific abstraction. We are encouraged by the fact that our asymptotic complexity is lower than the above mentioned procedures by orders of magnitudes. Moreover, our implementation is also faster by a factor of 100 than the one reported in [2][1]. The MONA System [11] can be used to implement the operations in this paper. However, it has non-elementary complexity and is in our experience infeasible for program with trees.

Acknowledgements. We thank Noam Rinetzky and the anonymous CAV referees for many helpful comments.

[1] Our method also allows trees which are beyond the scope of [2].

References

1. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In VMCAI, pages 164–180, 2005.
2. J. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. Tech. Rep. TR-2005-19, Dept. of Comp. Sci., Univ. of BC, Canada, 2005.
3. G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In CONCUR, pages 168–182, 2000.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In Symp. on Princ. of Prog. Lang., pages 269–282, New York, NY, 1979. ACM Press.
5. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In TACAS, pages 287–302, 2006.
6. R. Ghiya and L. Hendren. Putting pointer analysis to work. In Symp. on Princ. of Prog. Lang., New York, NY, 1998. ACM Press.
7. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In CAV, 1997.
8. L. Hendren. Parallelizing Programs with Recursive Data Structures. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.
9. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification via structure simulation. In Proc. Computer-Aided Verif., pages 281–294, 2004.
10. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, Program Flow Analysis: Theory and Applications, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.
11. N. Klarlund and A. Møller. MONA Version 1.4 User Manual. BRICS Notes Series NS-01-1, Dept. of Comp. Sci., Univ. of Aarhus, January 2001.
12. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In POPL, 2006.
13. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In ESOP, pages 124–140, 2005.
14. T. Lev-Ami, N. Immerman, T. W. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In CADE, pages 99–115, 2005.
15. T. Lev-Ami, N. Immerman, and M. Sagiv. Fast and precise abstraction for shape analysis. Technical Report TR-2006-01-001221, Tel-Aviv Univ., 2006. Available at http://www.cs.tau.ac.il/∼tla/2006/papers/TR-2006-01-001221.pdf.
16. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In Static Analysis Symp., pages 280–301, 2000.
17. R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In SAS, pages 265–279, 2004.
18. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In VMCAI, pages 181–198, 2005.
19. T. Reps, A. Loginov, and M. Sagiv. Semantic minimization of 3-valued propositional formulae. In LICS, pages 40–54, 2002.
20. Cananda Sable Research Group, McGill University. Soot: a java optimization framework. Available at: http://www.sable.mcgill.ca/soot/.
21. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. Trans. on Prog. Lang. and Syst., 2002.
22. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In TACAS, pages 530–545, 2004.