Partially Disjunctive Heap Abstraction

Roman Manevich[1], Mooly Sagiv[1], G. Ramalingam[2], and John Field[2]

[1] Tel Aviv University {rumster,msagiv}@tau.ac.il
[2] IBM T.J. Watson Research Center {jfield,rama}@watson.ibm.com

Abstract. One of the continuing challenges in abstract interpretation is the creation of abstractions that yield analyses that are both *tractable* and *precise enough* to prove interesting properties about real-world programs. One source of difficulty is the need to handle programs with different behaviors along different execution paths. Disjunctive (powerset) abstractions capture such distinctions in a natural way. However, in general, powerset abstractions increase space and time costs by an exponential factor. Thus, powerset abstractions are generally perceived as very costly.

In this paper, we partially address this challenge by presenting and empirically evaluating a new heap abstraction. The new heap abstraction works by merging shape descriptors according to a partial isomorphism similarity criteria, resulting in a partially disjunctive abstraction.

We implemented this abstraction in TVLA—a generic system for implementing program analyses.We conducted an empirical evaluation of the new abstraction and compared it with the powerset heap abstraction. The experiments show that analyses based on the partially disjunctive heap abstraction are as precise as the ones based on the powerset heap abstraction. In terms of performance, analyses based on the partially disjunctive heap abstraction are often superior to analyses based on the powerset heap abstraction. The empirical results show considerable speedups, up to 2 orders of magnitude, enabling previously non-terminating analyses, such as verification of the Deutsch-Schorr-Waite scanning algorithm, to terminate with no negative effect on the overall precision. Indeed, experience indicates that the partially disjunctive shape abstraction improves performance across all TVLA analyses uniformly, and in many cases is essential for making precise shape analysis feasible.

1 Introduction

One of the continuing challenges in abstract interpretation [3] is the creation of abstractions that yield analyses that are both *tractable* and *precise enough* to prove interesting properties about real-world programs. In this paper we partially address this challenge by presenting and empirically evaluating a new heap abstraction, i.e., an abstraction for the (potentially unbounded) dynamically allocated storage manipulated by programs (e.g., see [7, 9, 2, 8, 16, 14, 15]). Heap abstractions are of fundamental importance to static analysis and verification of programs written in modern languages. Heap abstractions have been used, for instance, in the context of shape analysis (e.g., for proving that a program fragment preserves certain tree structure invariants), as well as in verifying that a client program satisfies certain conformance constraints for the correct usage of a library.

We present our abstraction in the context of the parametric abstract interpretation framework of [15], which is based on the idea of representing program states using 3-valued logical structures. While it is very natural to view the abstraction we present as a heap abstraction, it can be used for abstracting other domains as well.

The TVLA framework presented in [15] uses a disjunctive (powerset) heap abstraction: the abstract value at every program point is a *set* of shape descriptors (of bounded size) and set union is used as the join operation. In particular, this abstraction does not attempt to combine (or merge) different shape descriptors into one and relies on the fact that there are only finitely many shape descriptors (as they are of bounded size). This leads to powerful and sophisticated analyses for proving interesting program properties but is usually too expensive to be applied to real-world programs. (The number of distinct shape descriptors is doubly exponential in the size of the program in the worst case.)

The heap abstractions most commonly used in practice, especially when scalability is important, tend to be *single-shape* heap abstractions, which use a single shape descriptor to describe all possible program states at a program point [9, 2, 14]. The current TVLA implementation provides options to utilize such single-shape heap abstractions. However, our experience has been that for the kind of applications that we have used TVLA for (mostly verification problems), the single-shape abstraction tends to be imprecise and causes a number of "false alarms" (i.e., verification fails for correct programs). Hence, this abstraction is not widely used by TVLA users. (A detailed discussion of the single-shape abstractions is beyond the scope of this paper, because of the complexity of formalizing the single-shape abstractions within the framework of 3-valued-logic.)

This paper presents a *partially disjunctive* heap abstraction which, in our experience, is significantly more efficient than the powerset heap abstraction, but has turned out to be precise enough for all the applications we have experimented with. Indeed, this abstraction has turned out to be the abstraction of choice for all TVLA users. The main idea behind this abstraction is to reduce the set of shape descriptors arising at a program point by merging "similar" shape descriptors but keeping "dissimilar" shape descriptors apart.

1.1 Running Example

Figure 1 shows a method implementing the mark phase of a mark-and-sweep garbage collector. The challenge here is to show that this procedure is partially correct, i.e., to establish that "upon termination, an element is marked if and only if it is reachable from the root." This simple program serves as a running example in this paper.

The partial correctness of this program was established using abstract interpretation in [13]. This abstract interpretation was created using TVLA—a generic system for implementing program analyses [10]. The default implementation of TVLA uses the powerset heap abstraction. Verification of the above property using the powerset heap abstraction took 584 cpu seconds and generated $189,772$ different shape descriptors—definitely too many for such a simple program and simple property. The situation is worse for verifying a similar property for an implementation of the Deutsch-Schorr-

```
// @Ensures marked == REACH(root)
void mark(Node root, NodeSet marked) {
   Node x;
   if (root != null) {
      NodeSet pending = new NodeSet();
      pending.add(root);
      marked.clear();
      while (!pending.isEmpty()) {
         x = pending.selectAndRemove();
         marked.add(x);
         if (x.left != null)
            if (!marked.contains(x.left))
               pending.add(x.left);
         if (x.right != null)
            if (!marked.contains(x.right)
               pending.add(x.right);
      }
   }
}
```

Fig. 1. A simple Java-like implementation of the mark phase of a mark-and-sweep garbage collector

Waite scanning procedure [11]. This verification took 4 hours when the powerset heap abstraction was used.

Powerset heap abstractions are costly since they may distinguish between too many shape descriptors, which may not be necessary in order to verify program properties. In this paper, we define a partially disjunctive heap abstraction, which is coarser than the powerset heap abstraction. The main idea is to reduce the set of shape descriptors arising at a program point by merging "similar" shape descriptors. In the mark example, verification using the partially disjunctive heap abstraction took 3 cpu seconds and generated $1,133$ shape descriptors—a two orders of magnitude improvement over verification using the powerset heap abstraction—with the same precision. Similarly, the verification of an implementation of the Deutsch-Schorr-Waite scanning procedure terminated successfully in 158 cpu seconds using the partially disjunctive heap abstraction.

1.2 Main Results

A New Abstraction. We define a new heap abstraction, which we refer to as the *partial-isomorphism* heap abstraction. The new abstraction is coarser than the powerset heap abstraction and yet keeps certain shape descriptors apart. Our abstraction is parametric. It allows the user to specify which heap properties are of importance for a given analysis, and this guides the abstraction in determining which shape descriptors are merged together.

Robust Implementation. We implemented our abstraction in TVLA. This abstraction has turned out to be the abstraction of choice for all TVLA users (e.g., see [19]). We believe that it is simple enough to be implemented in other systems besides TVLA (e.g., [17]).

Empirical Evaluation. We empirically evaluated our abstraction by comparing it with the powerset heap abstraction. In the largest benchmark, `SQLExecuter`, powerset heap abstraction did not terminate within $20,000$ cpu seconds. In contrast, the new abstraction took $9,673$ cpu seconds and proved correct usage of JDBC objects and absence of null-dereferences.

1.3 Outline

In Section 2, we give an overview of 3-valued-logic based program analysis. In Section 3, we describe the partial-isomorphism heap abstraction. In Section 4, we provide an empirical evaluation of the partial-isomorphism heap abstraction and powerset heap abstraction. In Section 5, we outline several other heap abstractions that we are investigating as ongoing work. In Section 6, we discuss related work.

2 3-valued Shape Analysis Primer

We now present an overview of *first order transition systems* (FOTS), the formalism underlying the parametric analysis framework of [15]. FOTS may be thought of as an imperative language built around an expression sub-language based on first-order logic with transitive closure.

Concrete Program Configurations

In FOTS, program states are represented using 2-valued logical structures.

Definition 1. *A 2-valued logical structure over a set of predicates P is a pair $C^\natural = \langle U^\natural, I^\natural \rangle$ where:*

- *$U^\natural$ is the universe of the 2-valued structure.*
- *$I^\natural$ is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in P$ of arity k, $I^\natural(p) : {U^\natural}^k \to \{0, 1\}$.*

In the context of shape analysis, a logical structure is used as a shape descriptor, with each individual corresponding to a heap-allocated object and predicates of the structure corresponding to properties of heap-allocated objects.

In the following, we use $p^{C^\natural}(v)$ as alternative notation for $I^\natural(p)(v)$, omitting the superscript $C^\natural$, when no confusion is likely. We denote the set of all 2-valued logical structures over a set of predicates P by 2-STRUCT$_P$. We will mostly assume that the set of predicates P is fixed and abbreviate 2-STRUCT$_P$ to 2-STRUCT.

Table 1. Predicates used to verify the running example

Predicates	Intended Meaning
$x(v)$	Does reference variable x point to object v?
$root(v)$	Does reference variable $root$ point to object v?
$\texttt{left}(v_1, v_2)$	Does field `left` of object v_1 point to object v_2?
$\texttt{right}(v_1, v_2)$	Does field `right` of object v_1 point to object v_2?
$r[root](v)$	Is object v heap-reachable from reference variable $root$?
$set[marked](v)$	Is object v a member of the $marked$ set?
$set[pending](v)$	Is object v a member of the $pending$ set?

Table 1 shows the predicates used to record properties of individuals for the analysis of our running example. A unary predicate $ref(v)$ holds when the reference (or pointer) variable `ref` points to the object v; in our example $\texttt{ref} \in \{x, root\}$. Similarly, a binary predicate $fld(v_1, v_2)$ records the value of a reference (or pointer-valued) field `fld`; in our example $\texttt{fld} \in \{\texttt{left}, \texttt{right}\}$. A unary predicate $set[s](v)$ holds when the object v belongs to the set s; in our example $s \in \{marked, pending\}$.

In this paper, program configurations (i.e., 2-valued logical structures) are depicted as directed graphs. Each individual of the universe is drawn as a node. A unary predicate $p(u)$, which holds for a node u, is drawn inside the node u. If a unary predicate represents a reference variables it is shown by having an arrow drawn from its name to the node pointed by the variable. A binary predicate $p(u_1, u_2)$ which evaluates to 1 is drawn as directed edge from u_1 to u_2 labelled with p.

Figure 2(a) shows a concrete configuration arising at the exit label of the mark procedure, where all the individuals that are reachable from $root$ are marked, as indicated by the value of the $set[marked]$ predicate. The individuals represented by the empty nodes correspond to garbage objects.

Operational Semantics

In FOTS, program statements are modelled by *actions* that specify how statements transform an incoming logical structure into an outgoing logical structure. This is done primarily by defining the values of the predicates in the outgoing structure using first-order logical formulae with transitive closure over the incoming structure [15].

Abstract Program Configurations

We now describe the abstractions used to create a finite (bounded) representation of a potentially unbounded set of 2-valued structures (representing heaps) of potentially unbounded size. The abstractions we use are based on 3-valued logic [15], which extends boolean logic by introducing a third value $1/2$, denoting values that may be 0 or 1. In particular, we utilize the partially ordered set $\{0, 1, 1/2\}$ with the join operation $\sqcup$, defined by $x \sqcup y = x$ if $x = y$ and $1/2$ otherwise.

Definition 2. *A 3-valued logical structure over a set of predicates P is a pair $C = \langle U, I \rangle$ where:*

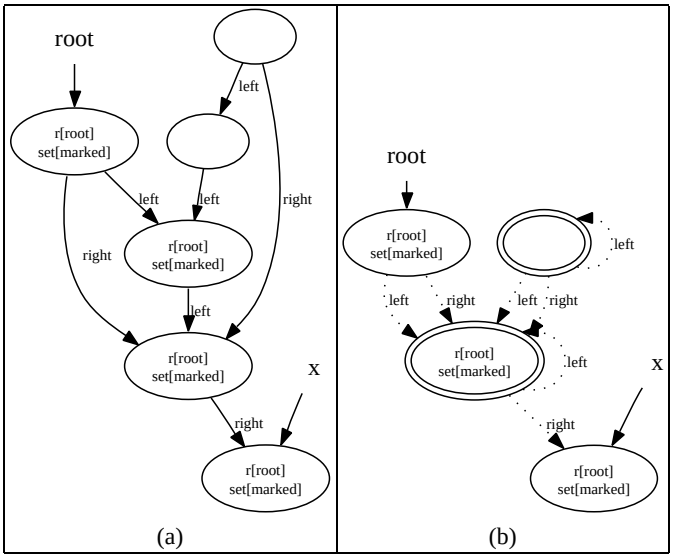

Fig. 2. (a) A concrete program configuration arising at the exit label of the mark procedure, where all non-garbage nodes have been marked; (b) An abstract program configuration that approximates the concrete configuration in (a)

- *U is the universe of the 3-valued structure.*
- *I is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in P$ of arity k, $I(p) : U^k \to \{0, 1, 1/2\}$.*

A 3-valued logical structure can be used as an abstraction of a larger 2-valued logical structure. This is achieved by letting an abstract configuration (i.e., a 3-valued logical structure) include *summary individuals*, i.e., an individual which corresponds to one or more individuals in a concrete configuration represented by that abstract configuration. In the rest of the paper, we assume that the set of predicates P includes a distinguished unary predicate *sm* to indicate if an individual is a summary individual.

In this paper, 3-valued logical structures are also depicted as directed graphs, where binary predicates with $1/2$ values are shown as dotted edges and summary individuals are shown as double-circled nodes.

We denote the set of all 3-valued logical structures over a set of predicates P by 3-STRUCT$_P$, usually abbreviating it to 3-STRUCT. We define a preorder on structures, denoted by $\sqsubseteq$, based on the concept of *embedding*.

Definition 3. *Let S and S' be two structures and let $f : U^S \to U^{S'}$ be surjective. We say that f* **embeds** *S in S' (denoted by $S \sqsubseteq^f S'$) if (i) for every predicate p (including sm) of arity k, and every $k - tuple$ of individuals $u_1, \ldots, u_k \in U^S$,*

$$p^S(u_1, \ldots, u_k) \sqsubseteq p^{S'}(f(u_1), \ldots, f(u_k)) \tag{1}$$

and (ii) for all $u' \in U^{S'}$

$$(|\{u \mid f(u) = u'\}| > 1) \sqsubseteq sm^{S'}(u') \tag{2}$$

We say that S **can be embedded in** S' *(denoted by* $S \sqsubseteq S'$*) if there exists a function* f *such that* $S \sqsubseteq^f S'$.

Bounded Program Configurations

Note that the size of a 3-valued structure is potentially unbounded and that 3-STRUCT is infinite. The abstractions studied in this paper rely on a fundamental abstraction function for converting a potentially unbounded structure (either 2-valued or 3-valued) into a bounded 3-valued structure, which we define now. This abstraction function $\alpha_{blur[A]}$ is parameterized by a special set of unary predicates A referred to as the *abstraction* predicates.

Let A be a set of unary predicates. An individual u_1 in a structure S_1 is said to be *A*-compatible to an individual u_2 in a structure S_2 iff for every predicate $p \in A$, $p^{S_1}(u_1) \sqsubseteq p^{S_2}(u_2)$ or $p^{S_2}(u_2) \sqsubseteq p^{S_1}(u_1)$. (Recall that the partial order $\sqsubseteq$ on $\{0, 1, 1/2\}$ is defined by $x \sqsubseteq y$ iff $x = y$ or $y = 1/2$.)

A 3-valued structure is said to be *A*-bounded if no two different individuals in its universe are *A*-compatible. A structure that is *A*-bounded can have at most $2^{|A|}$ individuals. We denote the set of all 3-valued *A*-bounded structures over a set of predicates by B-STRUCT$_{P,A}$, and, as usual, omit the subscripts when no confusion is likely.

The abstraction function β_{blur} : 3-STRUCT $\rightarrow$ B-STRUCT, which converts a (potentially unbounded) 3-valued structure into a bounded 3-valued structure, is defined as follows: we obtain an *A*-bounded structure from a given structure by merging all pairs of *A*-compatible individuals. $\beta_{blur}(\langle U_1, I\rangle) = \langle U_2, J\rangle$, where U_2 is the set of *A*-compatible equivalence classes of U_1, and the interpretation J is defined by:

$$\begin{array}{lll} p^J(c_1, \ldots, c_k) & = \sqcup_{u_1 \in c_1, \ldots, u_k \in c_k} p^I(u_1, \ldots, u_k) & \text{for } p \neq sm \\ sm^J(c) & = 1/2 & \text{if } |c| > 1 \\ sm^J(c) & = sm^I(u) & \text{if } c = \{u\}\ . \end{array}$$

Figure 2(b) shows an *A*-bounded structure obtained from the structure in Figure 2(a) with $A = \{x, root, r[root], set[marked], set[pending]\}$.

The abstraction function β_{blur} serves as the basis for abstract interpretation in TVLA. In particular, it serves as the basis for defining various different abstractions for the (potentially unbounded) *set of* 2*-valued logical structures* that arise at a program point.

2.1 Powerset Heap Abstraction

This abstraction is based on the fact that there can only be a finite number of bounded structures that are not *isomorphic* to one another. (Two structures are isomorphic when there is a bijection between their universes that preserves all predicate values.) The powerset abstraction function operates by bounding 2-valued structures with respect to a subset of the unary predicates, and removing duplicates (isomorphic structures).

For the sake of simplicity we will work with *canonic* bounded structures. Note that the individuals of an *A*-bounded structure are uniquely identified by the set of values of the predicates in A; we refer to such a set of predicate values as the individual's *canonical name*. For example, the individual pointed by $root$ in Figure 2(b) has the canonical name $u_{\{x=0,root=1,r[root]=1,set[marked]=1,set[pending]=0\}}$. A canonic bounded structure is a bounded structure in which the individuals are identified by their canonical names. We refer to the set of all canonic bounded structures by CB-STRUCT$_{P,A}$. Note that for a given P and A, CB-STRUCT$_{P,A}$ is finite. The *canonic* abstraction function $\beta_{canonic}$: 2-STRUCT $\rightarrow$ CB-STRUCT is defined as follows: $\beta_{canonic}(S)$ is obtained by renaming the individuals of $\beta_{blur}(S)$, giving them canonic names.

The powerset heap abstraction function $\alpha_{pow} : 2^{\text{2-STRUCT}} \rightarrow 2^{\text{CB-STRUCT}}$ is defined by

$$\alpha_{pow}(XS) = \{\beta_{canonic}(S) \mid S \in XS\} \ .$$

3 The Partial-Isomorphism Heap Abstraction

The idea behind partial-isomorphism heap abstraction is fairly simple. The powerset heap abstraction keeps all the canonic bounded structures that arise at a program point separate. Single-shape heap abstraction merges all canonic bounded structures arising at a program point into one structure. The partial-isomorphism heap abstraction, in contrast, merges canonic bounded structures into one structure only when they have the same universe.

We say that a pair of canonic bounded structures are *universe congruent* iff the two structures have the same universe. Universe congruence induces an equivalence relation over sets of canonic bounded structures. This equivalence relation lets us define an abstraction function $\alpha_{pi} : 2^{\text{2-STRUCT}} \rightarrow 2^{\text{CB-STRUCT}}$ that merges all universe congruent structures. Given a set of canonic bounded structures *XS* with the same universe U, we define the merged structure $\bigsqcup XS = \langle U, I\rangle$ that has the same universe as all structures in *XS* and the following interpretation of predicates. For every predicate p of arity k and tuple of individuals $\langle u_1, \ldots, u_k\rangle \in U^k$:

$$p^{\bigsqcup XS}(u_1, \ldots, u_k) = \bigsqcup_{S \in XS} p^S(u_1, \ldots, u_k) \ .$$

We are now ready to define the partial-isomorphism heap abstraction function α_{pi}:

$$\alpha_{pi}(XS) = \left\{\bigsqcup C \mid C \subseteq \alpha_{pow}(XS) \text{ is a universe congruence equivalence class}\right\} \ .$$

Thus, partial-isomorphism heap abstraction is less precise than the powerset heap abstraction[3]. As the empirical results presented later show, the partial-isomorphism heap abstraction seems to work as well as (i.e., is as precise as) the powerset heap abstraction, *in practice*. The following propositions may help explain why.

[3] Here, precision is used in the sense of a Galois Connection between a pair of abstract domains.

Proposition 1. *If a pair of bounded structures S_1 and S_2 are universe congruent, then the merged structure $S_1 \bigsqcup S_2$ is the least bounded structure that approximates (embeds) both S_1 and S_2.*

When partial-isomorphism abstraction is applied to a pair of structures S_1 and S_2, there are two possibilities:

- Structures S_1 and S_2 are not universe congruent. In this case, the result of the abstraction is $\alpha_{pi}(\{S_1, S_2\}) = \{S_1, S_2\}$, which is the least upper-bound of the powerset abstraction—the most precise approximation of both structures.
- Structures S_1 and S_2 are universe congruent. In this case, the result of the abstraction is $\alpha_{pi}(\{S_1, S_2\}) = S_1 \bigsqcup S_2$, which is the most precise upper bound among all (singleton sets of) bounded structures.

Proposition 2. *Partial-isomorphism heap abstraction preserves the values of abstraction predicates.*

In other words, partial-isomorphism heap abstraction only loses the same kind of distinctions that can also be lost by β_{blur}—values of non-abstraction predicates.

In terms of worst-case complexity, partial-isomorphism heap abstraction has the same complexity as powerset heap abstraction—doubly-exponential in the number of abstraction predicates. This is due to the number of sets of canonical names, which is the dominant factor in the worst-case complexity. However, partial-isomorphism heap abstraction can save an exponential factor due to binary predicates, which is the dominant factor in many cases, in practice.

3.1 Illustrating Example

To illustrate the operation of partial-isomorphism heap abstraction, consider the abstract program configuration shown in Figure 2(b) and the abstract program configuration shown in Figure 3(a). Both configurations represent cases where all of the non-garbage nodes have been marked and non-garbage nodes have not been marked, i.e., the program property we want to verify holds for those configurations. The difference between the configurations is in the position of the node pointed by `x` in the part of the heap that has been marked. In this case, the partial-isomorphism heap abstraction results in the structure shown in Figure 3(b), which ignores the precise position of the node pointed by `x` inside the part of the heap that was marked.

The mark program non-deterministically selects an object and removes it from the pending set. This non-determinism allows many different ways of traversing the set of objects reachable from `root`, which results in many different abstract program configurations that sustain the program property we want to verify and only differ by values of binary predicates. Partial-isomorphism heap abstraction ignores the values of the binary predicates, but keeps precise the overall property for an abstract configuration of having sets of nodes with the same garbage/non-garbage and mark/unmarked properties. This allows the analysis to merge many similar structures without losing the information needed to prove the partial correctness of the mark program.

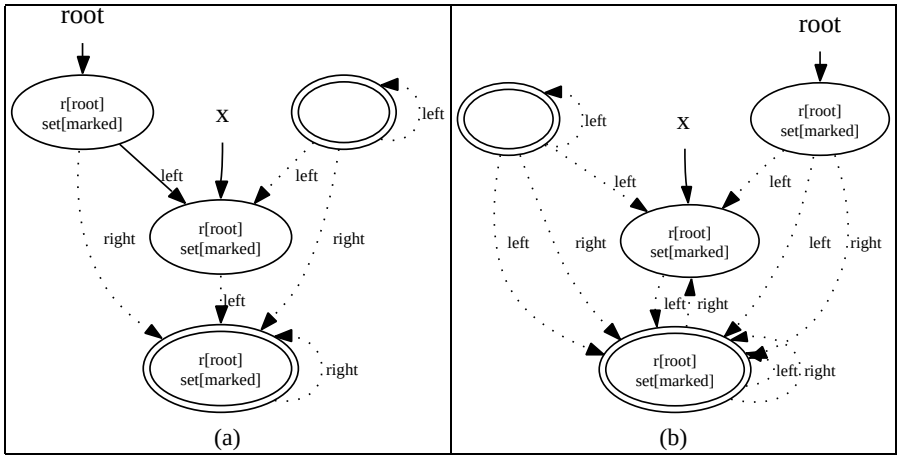

Fig. 3. (a) An abstract program configuration arising at the exit label of the mark procedure, where all non-garbage nodes have been marked and `x` points to a node adjacent to `root`; (b) The result of merging the structure in (a) and the structure in Figure 2(b)

4 Implementation and Empirical Evaluation

We implemented the partial-isomorphism abstraction described in the previous section in TVLA, and the implementation is publicly available [10]. We applied it to verify various specification for the Java programs described in Table 2. To translate Java programs and their specifications to TVP (TVLA's input language), we used a front-end for Java, which is based on the Soot framework [18]. For all benchmarks, we checked the absence of null dereferences in addition to the properties described in Table 2. Our specifications include correct usage of JDBC objects, correct usage of Java I/O streams, correct usage of Java collections and iterators, and additional small but interesting specifications.

The experiments were conducted using TVLA version 2, running with SUN's JRE 1.4, on a 1 GHZ Intel Pentium Processor machine with 1.5 GB RAM. We optimized for precision and simplicity by using TVLA's Focus and Coerce operations in all benchmarks. We compared partial isomorphism to the full powerset abstraction in terms of time and space performance and precision.

The results of the analyses are shown in Table 3. In all the benchmarks the analysis based on the partial-isomorphism heap abstraction achieved the same precision as the analysis based on the powerset heap abstraction, and other TVLA users reported the same phenomena. In all but one example, the analysis based on partial-isomorphism heap abstraction achieved significant performance improvements.

Table 2. Benchmarks and properties used for comparing the analysis based on powerset heap abstraction with the analysis based on partial-isomorphism heap abstraction. Treeness means preservation of tree structure invariants

Benchmark	**Description**	**Property**
GC.mark	Figure 1	Partial correctness
DSW	Deutsch-Schorr-Waite	Partial correctness of tree scanning + Treeness
ISPath	Input streams	Correct usage of Java IOStreams
InputStream5	Input stream holders	Correct usage of Java IOStreams
InputStream5b	Input stream holders with error	Correct usage of Java IOStreams
InputStream6	Input stream holders	Correct usage of Java IOStreams
SQLExecutor	A JDBC framework	Correct usage of JDBC objects
KernelBench.1	CMP benchmark [12]	Absence of concurrent modification exceptions
InsertSorted	Insertion into sorted trees	Tree sortedness + Treeness
DeleteSorted	Deletion from sorted trees	Tree sortedness

Table 3. Time, space and number of errors measurements. Rep. Err. is the number of errors reported by the analysis, and Act. Err. is the number of errors that indicate real problems. Time and space measurements for non-terminating benchmarks are prefixed with $>$ to indicate the measurements taken when the analysis timed out. The number of reported errors is the same for both the analysis based on the powerset heap abstraction and the analysis based on partial-isomorphism heap abstraction on all (terminating) benchmarks. For benchmarks that did not terminate with the powerset heap abstraction, the numbers are taken from the analysis based on partial-isomorphism heap abstraction

Benchmark	**Time in seconds**		**Space in Mb.**		**Rep. Err. / Act. Err.**
	Powerset	**Partial iso.**	**Powerset**	**Partial iso.**	
GC.mark	584	3	56	1.4	0/0
DSW	14,364	157	116.3	5.6	0/0
ISPath	79	79	2.8	2.9	0/0
InputStream5	4,530	1,706	14.0	11.9	1/0
InputStream5b	3,492	1,394	9.8	9.1	1/0
InputStream6	15,558	3,929	23.6	15.9	1/0
SQLExecutor	$>$20,000	9,673	$>$109.3	104.8	0/0
KernelBench.1	7,393	5,355	13.3	10.8	1/1
InsertSorted	264	37	4.5	2.4	0/0
DeleteSorted	$>$20,000	3,271	$>$62.6	21.8	0/0

4.1 Implementation Independent Results

Although the results shown in Table 3 measure the time and space consumption of analyses using different abstractions, they are also influenced by the various implementation details of the abstractions.

In Table 4, we supply implementation independent measurements. We measured the total number of abstract configurations generated by the analysis and the maximal number of abstract configurations that exist in the transition system at any given time during the analysis. The total number of abstract configurations and the maximal number of abstract configurations are always the same with the powerset heap abstraction, since structures are only accumulated in the transition system. For the partial-isomorphism heap abstraction, the maximal number of abstract configurations is often lower than the total number of abstract configurations, indicating that structures discovered in different iterations were merged together.

The results show a consistency between the improvements in time and space performance of the partial-isomorphism heap abstraction, relative to the powerset heap abstraction, and the reduced number of abstract configurations.

Table 4. Implementation independent measurements. Total #structs is the total number of abstract configurations that arose during the analysis, and Max #structs is the maximal number of abstract configurations that existed in the transition system at any time during the analysis. The results of non-terminating benchmarks are prefixed with $>$ to indicate the measurements taken when the analysis timed out

Benchmark	**Total #structs**		**Max #structs**	
	Powerset	**Partial iso.**	**Powerset**	**Partial iso.**
GC.mark	189,772	1,133	189,772	748
DSW	320,387	6,480	320,387	2,986
ISPath	2,168	2,168	2,168	2,168
InputStream5	8,164	3,366	8,164	2,204
InputStream5b	5,973	2,598	5,973	1,729
InputStream6	24,461	6,678	24,461	4,411
SQLExecutor	>8,824	4,107	>8,824	2,164
KernelBench.1	12,594	9,296	12,594	5,748
InsertSorted	7,487	1,318	7,487	905
DeleteSorted	>158,780	30,386	>158,780	25,673

5 Extensions and Future Work

The partial-isomorphism heap abstraction has so far performed quite satisfactorily in our experience with TVLA. However, we cannot assume that this will always be adequate. Analysis and verification of larger programs may require more aggressive abstractions, while in some cases we may require more precise abstractions. In this section

we describe various other abstractions that may be of value. We are currently in the process of evaluating the effectiveness of some of the abstractions described below.

Parametric Partial Isomorphism

We now present a parametric abstraction that includes both the powerset heap abstraction and the partial-isomorphism heap abstraction as special cases.

Definition 4. *We say that a pair of bounded structures $S_1 = \langle U_1, I_1 \rangle$ and $S_2 = \langle U_2, I_2 \rangle$ are* ***partially isomorphic*** *with respect to a set of predicates R, denoted by $S_1 \equiv_R S_2$, iff there exists a bijection $f^{pi} : U_1 \rightarrow U_2$, such that, for every predicate $p \in R$ of arity k and tuple of nodes $\langle u_1, \ldots, u_k \rangle \in U_1^k$, the following holds:*

$$p^{S_1}(u_1, \ldots, u_k) = p^{S_2}(f^{pi}(u_1), \ldots, f^{pi}(u_k)) \ .$$

Note that $\equiv_R$ is an equivalence relation among 3-valued structures. Given any set of predicates R that includes the set of all abstraction predicates A, we define an abstraction function $\alpha_{pi[R]} : 2^{\text{2-STRUCT}} \rightarrow 2^{\text{CB-STRUCT}}$ as follows:

$$\alpha_{pi[R]}(XS) = \left\{ \bigsqcup C \mid C \subseteq \alpha_{pow}(XS) \text{ is a } \equiv_R \text{ equivalence class} \right\} \ .$$

This function defines a whole family of abstractions. Further, $\alpha_{pow} = \alpha_{pi[P]}$ (where P is the set of all predicates) is the most precise among this family of abstractions, and $\alpha_{pi} = \alpha_{pi[A]}$ is the least precise among this family of abstractions.

The reason we restrict ourselves to sets R that contain the set of all abstraction predicates A is the following. If R includes A, then for any two $\equiv_R$-equivalent bounded structures, the bijection between the universes of the two structures that preserves the values of predicates in R is uniquely determined, and this bijection is used to determine which individuals should be "merged" together.

This parametric definition allows users to choose abstractions in a more fine-grained fashion, by specifying the set of predicates R. The parametric abstraction could also be used by an appropriate iterative refinement technique, which starts with $R = A$ and iteratively adds predicates to R, until a sufficiently precise abstraction is obtained or $R = P$.

Deflating Reductions

Deflating reductions can potentially yield performance improvements without a loss of precision. A very simple deflating reduction is the following: consider a set of 3-valued structures X containing structures S_1 and S_2, such that $S_1 \sqsubseteq S_2$. Clearly, the set $X' = X - \{S_1\}$ is semantically equivalent to X, and removing S_1 involves no loss of precision (even when the abstract transformer that is used is not the best). This reduction is referred to as "non-redundancy" in [1]. Making this reduction feasible requires testing for the partial order relation over 3-valued structures, which can be done

in polynomial time for bounded 3-valued structures. The key question with this reduction is whether the subsequent (performance) benefits of doing the reduction outweigh extra cost of performing the reduction. Our initial experience shows that this reduction is worth using. This reduction transforms TVLA's preorder over sets of 3-valued structures into a proper (Hoare powerdomain) partial ordering.

6 Related Work

A substantial body of literature exists on abstractions for various different domains and for creating new abstractions from existing abstractions. The distinguishing aspect of our work is its focus on heap abstractions and its focus on an empirical evaluation of the effectiveness of the proposed heap abstraction.

Function Space Domain Construction. Function space domain construction is one way of creating abstractions that are "partly disjunctive". Examples of previous work using such a domain construction include [5], where the abstraction is composed of two components—a lattice of symbolic access paths and a parametric numerical lattice. In this abstraction, abstract elements with the same symbolic access path component are merged by joining the numerical lattice component. The ESP system [4] also utilizes a similar function space domain construction, but not for heap abstractions.

Least Disjunctive Basis. In [6], a technique is defined for obtaining the "least disjunctive basis", which is the most abstract domain inducing the same disjunctive completion as another domain. Unfortunately, this may result in larger sets of abstract elements, as abstract elements are substituted by sets of other abstract elements, causing inflation.

Deflating Operators and Widening Operators. In [1], different widening operators and congruence relations are considered for the powerset polyhedra domain, and in more general settings.

Acknowledgements

The authors wish to thank Alexey Loginov for supplying us the tree benchmarks and the DSW benchmark, and Eran Yahav for supplying us the IOStream benchmarks, the KernelBench.1 benchmark and the SQLExecutor benchmark. The authors also wish to thank Noam Rinetzky and Greta Yorsh for supplying helpful comments on earlier drafts of this paper.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. In B. Steffen and G. Levi, editors, *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 135–148, Venice, Italy, 2003. Springer-Verlag, Berlin.

2. D. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 296–310, New York, NY, 1990. ACM Press.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 238–252, New York, NY, 1977. ACM Press.
4. M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Jan. 2002.
5. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 230–241, New York, NY, 1994. ACM Press.
6. R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Science of Computer Programming*, 32(1-3):177–210, 1998.
7. N. Jones and S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.
8. J. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, Univ. of Calif., Berkeley, CA, May 1989.
9. J. Larus and P. Hilfinger. Detecting conflicts between structure accesses. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 21–34, New York, NY, 1988. ACM Press.
10. T. Lev-Ami and M. Sagiv. TVLA: A framework for implementing static analyses. In J. Palsberg, editor, *Proc. Static Analysis Symp.*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer-Verlag, 2000. Available from http://www.cs.tau.ac.il/∼tvla/.
11. G. Lindstrom. Scanning list structures without stacks or tag bits. *Inf. Process. Lett.*, 2(2):47–51, June 1973.
12. G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 83–94. ACM Press, June 2002.
13. T. Reps, M. Sagiv, and R. Wilhelm. Automatic verification of a simple mark and sweep garbabge collector. Presented in the 2001 University of Washington and Microsoft Research Summer Institute, Specifying and Checking Properties of Software, http://research.microsoft.com/specncheck/, 2001.
14. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Prog. Lang. Syst.*, 20(1):1–50, Jan. 1998.
15. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
16. J. Stransky. A lattice for abstract interpretation of dynamic (lisp-like) structures. *Information and Computation*, 101(1):70–102, Nov. 1992.
17. T.Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *Static Analysis Symposium*, pages 69–84, 2002.
18. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.
19. E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstraction. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, New York, NY, 2004. ACM Press. To Appear.