Combining Shape Analyses by Intersecting Abstractions

Gilad Arnold[1], Roman Manevich[2⋆], Mooly Sagiv[2], and Ran Shaham[3]

[1] University of California, Berkeley arnold@eecs.berkeley.edu
[2] Tel Aviv University {rumster,msagiv}@tau.ac.il
[3] ran.shaham@gmail.com

Abstract. We consider the problem of computing the intersection (meet) of heap abstractions.This problem is useful, among other applications, to relate abstract memory states computed by forward analysis with abstract memory states computed by backward analysis. Since dynamically allocated heap objects have no static names, relating objects computed by different analyses cannot be done directly. We show that the problem of computing meet is computationally hard. We describe a constructive formulation of meet based on certain relations between abstract heap objects. The problem of enumerating those relations is reduced to finding constrained matchings in graphs. We implemented the algorithm in the TVLA system and used it to prove temporal heap properties of several small Java programs, and obtained empirical evidence showing the effectiveness of the meet algorithm.

1 Introduction

This research is motivated by the need to approximate temporal properties of programs manipulating dynamically allocated data structures. For example, statically identifying a point in the program after which a list element will never be accessed and thus can be deallocated. As it is undecidable, in general, to prove interesting properties about programs with dynamic memory allocation with pointers and destructive updates, the use of abstract interpretation [2] to compute an over-approximation of a program's operational semantics is a fundamental practice underlying this work. Thus, while proving some correct program properties may fail, every proved property is assured to hold.

We are interested in inferring *persistent* [6] temporal properties of heaps. These are properties that continuously hold from a given point in the trace. Inferring persistent temporal properties is naturally done in two phases, where the first phase over-approximates the shapes of the data structures using forward analysis starting at the entry node, and the second phase computes heap liveness using a backward analysis stating at the exit node. Notice that this generalizes the process of computing scalar liveness in compilers in which the first phase is unnecessary in the absence of pointers and arrays. We call this approach *Phased Bidirectional Analysis*.

⋆ This research was supported in part by the Clore Fellowship Programme.

The problem of integrating the forward phase with the backward phase is challenging since the exact memory locations are lost by the abstraction. Therefore, this paper addresses the problem of computing the intersection of heap abstractions. When applied to a set of elements of some abstract domain (lattice), this operator—commonly referred to as *meet*—yields the greatest lower bound of the elements in the set. Specifically, for two heap abstractions, the corresponding meet is the set of common stores that are represented by both of its operands.

The main contributions of this paper are summarized as follows:

1. We prove that meet is computationally hard for the abstract domain of bounded structures (Theorem 3), which is used by the TVLA system, by showing a reduction from the problem of 3-colorability on graphs to deciding whether the output of meet is empty. This result is a bit surprising since structures in this domain have unique "canonical names", which makes isomorphism checking and checking of embedding (subsumption) decidable in polynomial time.
2. We present a new algorithm to compute the meet of 3-valued structures. We define the concept of *correspondence* relations between abstract heap objects and explain how to compute meet from these relations. We then develop a strategy to find correspondence relations that manages to prune many of the irrelevant relations thus making the algorithm efficient in practice.
3. We have implemented the meet algorithm in TVLA—a system for generating program analysis from operational semantics [5]—and used it to implement a new analysis for detecting program locations where heap objects and reference fields become unused in Java programs. The information discovered by the analysis can be used to improve memory management. The analysis combines forward and backward information and proves to be precise enough for several small but interesting programs operating on list data structures. The empirical results shows that our analysis is precise enough to reclaim memory as soon as it becomes unneeded. Therefore, our algorithm can serve as a reference algorithm for compile-time garbage collection. Our experiments indicate that the heuristics used by the meet algorithm make it very effective in combining shape analysis; the time and space performance of the algorithm is typically related to the size of the input and output by a linear factor. However, our current prototype implementation is slow and was only applied to small programs.

Running Example. Fig. 1 shows a simple program in a Java-like language that prints the elements of a singly-linked list. This program serves as the running example in this paper. The goal of the analysis here is to discover the earliest points where reference variables and reference fields are no longer used. Specifically, we would like to find that: (a) reference variable `x` is never used after line 7 (this is rather trivial, since `x` does not appear later), and (b) that the reference field `n` of the object pointed-to by `y` is never used after line 10. The second fact

```
 [1] x = null;
 [2] while (...) {
 [3]   y = new SLL();
 [4]   y.val = ...;
 [5]   y.n = x;
 [6]   x = y;
     }
 [7] y = x;     // can insert "x = null;" here
 [8] while (y != null) {
 [9]   System.out.print(y.val);
[10]   t = y.n; // can insert "free y;" or "y.n = null;" here
[11]   y = t;
     }
```

Fig. 1: A program that creates a singly-linked list and traverses its elements

is more challenging to prove, as the object pointed-to by y is different on every iteration of the loop.

Outline. The rest of the paper is organized as follows. Section 2 gives an overview of program analysis of heap-manipulating programs using 3-valued logic. In Section 3 we explain how approximate temporal properties of heaps with meet. In Section 4, we present our algorithm for meet. Section 5 describes our experiments with an analyzer that infers compile-time garbage collection information in Java programs by using meet. Section 6 discusses related work.

All proofs, as well as detailed examples, appear in [1].

2 3-Valued Shape Analysis Overview

In this section we explain the representation of concrete program states and their abstractions, based on the parametric analysis framework of [7].

2.1 Concrete Program States

We represent concrete program states by 2-valued logical structures.

Definition 1. *A 2-valued logical structure over a vocabulary (set of predicates) $\mathcal{P}$ is a pair $S = \langle U, \iota \rangle$ where U is the universe of the 2-valued structure, and ι is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k, $\iota(p) : U^k \to \{0, 1\}$.*

In this paper, we assume that the set of predicates includes the binary predicate *eq*, and insist that it is interpreted as equality between individuals. Table 1 shows the predicates used to record properties of individuals for the shape analysis of the running example (forward phase).

We denote the set of all 2-valued logical structures over a set of predicates $\mathcal{P}$ by 2-STRUCT[$\mathcal{P}$]. In the sequel, we assume that the vocabulary $\mathcal{P}$ is fixed, and abbreviate 2-STRUCT[$\mathcal{P}$] to 2-STRUCT.

Table 1. Predicates used for shape analysis of the running example, and their meaning. The set $PVar$ stands for the set of reference variables {x,y,t}

Predicates	**Intended Meaning**
$eq(v_1, v_2)$	Is v_1 equal to v_2?
$\{x(v) : x \in PVar\}$	Does reference variable `x` point to object v?
$n(v_1, v_2)$	Does the `n` field of object v_1 point to object v_2?
$\{r_{x,n}(v) : x \in PVar\}$	Is v reachable from reference variable `x` along `n` fields?
$is(v)$	Do two or more fields of heap elements point to v?
$c_n(v)$	Is v on a directed cycle of `n` fields?

Concrete states (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 an individual u, appears next to the corresponding node. If a unary predicate represents a reference variable, it is shown by having an arrow drawn from its name to the node referenced by the variable. The binary predicate $n(u_1, u_2)$, which holds for a pair of individuals u_1 and u_2, is drawn as a directed edge from u_1 to u_2, and labeled n. The predicate eq is not drawn, since any two nodes are different and every node is equal to itself.

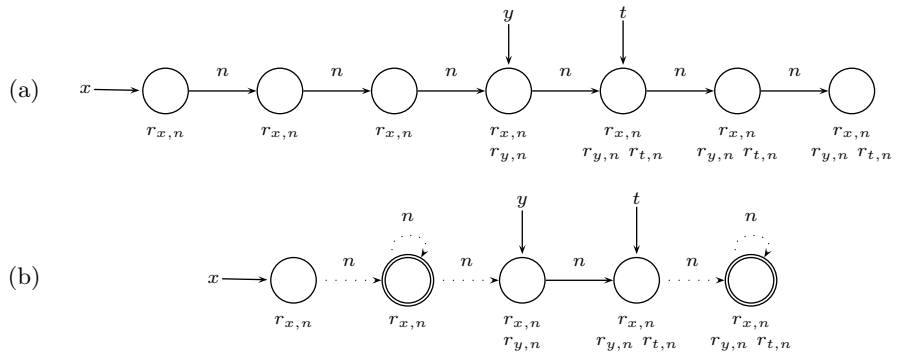

Fig. 2: (a) A concrete program state arising after the execution of the statement $t = y.n$; (b) An abstract program state approximating the concrete state in (a)

Fig. 2(a) shows a concrete program state arising after the execution of the statement $t = y.n$ on line 10 of the running example in Fig. 1.

2.2 Abstract Program States

The abstract program states we use are based on 3-valued logic [7], which extends boolean logic by introducing a third value 1/2, denoting values that may be either 0 or 1. In particular, we utilize the partially ordered set $\{0, 1, 1/2\}$ where $0 \sqsubseteq 1/2$ and $1 \sqsubseteq 1/2$, with the join operation $\sqcup$, defined by $x \sqcup y = x$ if $x = y$, and $x \sqcup y = 1/2$ otherwise.

Definition 2. *A 3-valued logical structure over a set of predicates $\mathcal{P}$ is a pair $S = (U, \iota)$ where U is the universe of the 3-valued structure, and ι is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k, $\iota(p) : U^k \to \{0, 1, 1/2\}$.*

An abstract state may include summary nodes*, i.e., an individual which corresponds to one or more individuals in a concrete state represented by that abstract state. A summary node u has $eq(u, u) = 1/2$, indicating that it may represent more than a single individual.*

Abstract states (3-valued logical structures) are also depicted as directed graphs, where unary predicates denoting reference variables, as well as binary predicates, with 1/2 values are shown as dotted edges. Summary individuals appear as double-circled nodes. A unary predicate that evaluates to 1/2 for a node is depicted by having $= 1/2$ next to the name of the predicate.

We denote the set of all 3-valued logical structures over a set of predicates $\mathcal{P}$ by 3-STRUCT[$\mathcal{P}$], and usually abbreviate it to 3-STRUCT.

We define a partial order on structures, denoted by $\sqsubseteq$, based on the concept of *embedding*.

Definition 3 (Embedding). *Let $S = (U, \iota)$ and $S' = (U', \iota')$ be two structures and let $f : U \to U'$ be a surjective function. We say that f* embeds *S in S', denoted $S \sqsubseteq^f S'$, if for every predicate $p \in \mathcal{P}^{(k)}$ and k individuals $u_1, \ldots, u_k \in U$,*

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

We say that S is embedded *in S', denoted $S \sqsubseteq S'$, if there exists a function f such that $S \sqsubseteq^f S'$. We also say that S'* approximates *S.*

The embedding order is used to define a concretization function for a single 3-valued structure S by $\sigma(S) = \{S' \in \text{2-STRUCT} \mid S' \sqsubseteq S\}$. The concretization of a set of 3-valued structures is defined by $\gamma(XS) = \bigcup_{S \in XS} \sigma(S)$.

The embedding order induces a Hoare preorder on sets of 3-valued structures.

Definition 4. *For sets of structures $XS_1, XS_2 \subseteq$ 3-STRUCT, $XS_1 \sqsubseteq XS_2$ if and only if $\forall S_1 \in XS_1 : \exists S_2 \in XS_2 : S_1 \sqsubseteq S_2$.*

In the following definition, we restrict sets of 3-valued structures by disallowing non-maximal structures. This ensures that the Hoare ordering is a proper partial ordering on the sets.

We are now ready to present the abstract domain which is considered for the construction of the meet algorithm.

Definition 5 (Core Abstract Domain). *The abstract domain $D_{\text{3-STRUCT}}$ consists of all sets of 3-valued structures that do not contain non-maximal structures, $\{XS \subset \text{3-STRUCT} \mid \forall S_1, S_2 \in XS : S_1 \sqsubseteq S_2 \implies S_1 = S_2\}$, with the same ordering as in Definition 4.*

2.3 Bounded Program States

Note that the size of a 3-valued structure is potentially unbounded and that 3-STRUCT is infinite. The abstractions studied in [7], and also used for the analysis in Section 5, rely on a fundamental abstraction function for converting a potentially unbounded structure—either 2-valued or 3-valued—into a bounded 3-valued structure.

A 3-valued structure is said to be *bounded* if for every two distinct individuals in its universe there exists a unary predicate p such that either $p^{S_1}(u_1) = 0$ and $p^{S_2}(u_2) = 1$ or $p^{S_1}(u_1) = 1$ and $p^{S_2}(u_2) = 0$.[4] We denote the set of all bounded 3-valued structures over a set of predicates $\mathcal{P}$ by B-STRUCT[$\mathcal{P}$]. The finite abstract domain $D_{\text{B-STRUCT}}$ is a sublattice of $D_{\text{3-STRUCT}}$, containing all sets of bounded structures that do not contain non-maximal structures.

The abstraction function $\beta^{\mathcal{P}}_{blur}$: 2-STRUCT[$\mathcal{P}$] $\rightarrow$ B-STRUCT[$\mathcal{P}$] converts a (potentially unbounded) 2-valued structure into a bounded 3-valued structure, by merging all individuals with the same values for all unary predicates. Namely, $\beta^{\mathcal{P}}_{blur}((U, \iota)) = (U', \iota')$, where U' is the set of equivalence classes in U of nodes with same values for all unary predicates, and the interpretation ι' of each predicate $p \in \mathcal{P}^{(k)}$ and k individuals $c_1, \ldots, c_k \in U'$ is given by

$$p^{S'}(c_1, \ldots, c_k) = \bigsqcup_{u_i \in c_i} p^S(u_1, \ldots, u_k) \ .$$

Fig. 2(b) shows a bounded structure obtained from the structure in Fig. 2(a).

The abstraction function β_{blur}, which is called *canonical abstraction*, serves as the basis for abstract interpretation in TVLA [5]. In particular, it serves as the basis for defining various different abstractions for the (potentially unbounded) set of 2-valued logical structures that may arise at a program point, by defining different sets of predicates. We also define the function α, which extends β_{blur} to sets of structures: $\alpha(XS) = \bigsqcup\{\beta_{blur}(S) \mid S \in XS\}$.[5]

3 Inferring Temporal Properties via Staged Bidirectional Analysis

Persistent temporal properties can be efficiently verified without explicitly representing traces. An example of such a property is liveness of reference variables and reference fields. A reference variable or reference field is said to be *dead* (i.e., not *live*) at a given program point if on every execution that goes through that point it is not used before being redefined.

[4] The notion of a bounded structure can be generalized by considering any subset of the set of unary predicates, as done in TVLA.

[5] The operator $\bigsqcup$ is the least upper bound operator in $D_{\text{B-STRUCT}}$.

The (possibly infinite) set of temporal properties is defined as the least fixed point of the following (not necessarily computable) system of equations:

$$\begin{aligned}
\overrightarrow{CS}_{entry} &= CS_{init} \\
\overrightarrow{CS}_{l_2} &= \left\{ S_{out} \,\middle|\, (l_1, l_2) \in E,\ S_{in} \in \overrightarrow{CS}_{l_1},\ (l_1, S_{in}) \overrightarrow{\rightsquigarrow} (l_2, S_{out}) \right\} \\
\overleftarrow{CS}_{exit} &= CS_{final} \cap \overrightarrow{CS}_{exit} \\
\overleftarrow{CS}_{l_1} &= \left\{ S_{in} \,\middle|\, (l_1, l_2) \in E,\ S_{out} \in \overleftarrow{CS}_{l_2},\ (l_1, S_{out}) \overleftarrow{\rightsquigarrow} (l_2, S_{in}) \right\} \cap \overrightarrow{CS}_{l_1} \ .
\end{aligned}$$

Here, it is assumed that the concrete 2-valued structures also record information on temporal properties that hold on program executions. The program is represented as a control flow graph, with entry and exit nodes *entry* and *exit*, respectively, and a set of control flow edges E. CS_{init} is the initial set of concrete stores at the entry location including all possible values associated with temporal properties. CS_{final} represents the set of states in which all temporal properties are set to their final values (that is, their values upon termination of the execution). We write $(l_1, S_{in}) \overrightarrow{\rightsquigarrow} (l_2, S_{out})$ to denote the transformation induced by the forward execution of the statement or condition at edge (l_1, l_2). Program conditions are interpreted according to the standard semantics. Note that the forward semantics sets values non-deterministically to the temporal properties predicates. We write $(l_1, S_{out}) \overleftarrow{\rightsquigarrow} (l_2, S_{in})$ to denote the transformation induced by the backward execution of the statement or condition at edge (l_1, l_2). This semantics sets the values of the changed temporal properties. Variables whose values are changed are updated non-deterministically.

The above system of equations does not necessarily terminate for programs with loops. Therefore, an upper approximation to this system is conservatively computed by representing sets of states using 3-valued structures. Extra predicates store values of tracked temporal properties. Moreover, the ability to define unary predicates allows tracking of an unbounded number of temporal properties. Both forward and backward executions are conservatively executed on 3-valued structures. However, as backward reasoning uses results obtained by the forward counterpart, it is considered a *secondary stage* taking place after the forward reasoning is complete. Finally, intersection ($\cap$) is over-approximated using meet ($\sqcap$).

3.1 Compile-Time GC Analysis

We now explain how compile-time garbage collection information can be computed using a phased bidirectional verification.

In particular, we are interested in identifying the first point in the trace where an object is not further used, and therefore may be safely deallocated by a `free` statement. Thus, the backward execution of a statement tracks the use of objects. Our analysis maintains the predicate $use(v)$ to track object future usage information.

An object v is denoted *used* in a statement or a condition at edge (l_1, l_2), if a reference expression e, that evaluates to v, is used for dereference at that statement. In such a case, the backward execution of the statement $(l_1, S_{out}) \overleftarrow{\rightsquigarrow} (l_2, S_{in})$

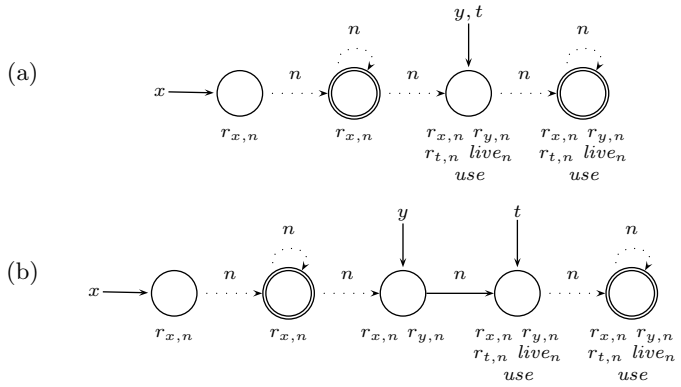

Fig. 3: 3-valued structures representing sets of program configurations, including heap object and reference field liveness, that arise (a) before the execution of the statement `t = y.n`; and (b) after it is executed

records in S_{in} the fact that v is used by setting $use(v)$ to 1. As mentioned, the forward execution of a statement non-deterministically sets values to $use(v)$.

Fig. 3(a) shows one of the structures that arise before the statement `t = y.n` at line 10 of Fig. 1, and Fig. 3(b) shows one of the structures that arise after that statement. The object referenced by `y` is still used before the statement, as $use(v)$ holds for the individual referenced by `y`. Nonetheless, the object referenced by `y` is not (further) used after that statement, as $use(v)$ does not hold for the individual referenced by `y`. Having verified that $use(v)$ does not hold for any individual v referenced by `y`, for all structures that may arise after the aforementioned statement, we conclude that `free y` may be inserted after the statement `t = y.n`, to deallocate the object referenced by `y`, as it is no longer used in the program. Moreover, since *for all structures* arising before that statement, the object referenced by `y` is still used, placing a `free y` after that statement will free the space referenced by `y` at the earliest possible time.

3.2 Assign-Null Analysis

Another application of phased bidirectional analysis is the computation of heap reference liveness, providing for compile-time optimization of runtime garbage collection effectiveness. For each object reference field, we identify whether it is *live* at any point in the trace, meaning that it may be used, prior to being redefined, after that point. We are interested in spotting points in the trace where a reference field becomes *dead*, and therefore may be assigned a `null` value, thus significantly reducing potential GC drag time [8]. Here, again, the backward execution of the statement tracks the uses (dereference) and redefinitions (assignment) of object fields. In particular, for each reference field f which is a

member of some object v, the predicate $live_f(v)$ is used to record future use and re-definition information (in our example f is n).

A reference field f of an object v is denoted *used* in a statement or a condition at edge (l_1, l_2) if an expression e—which is not an l-value—refers to the value of f. In this case, the backward execution of the statement $(l_1, S_{out}) \overleftsquigarrow (l_2, S_{in})$ sets $live_f(v)$ to 1. Otherwise, f is denoted *redefined* if it is being assigned a new value, namely, being referred to by an l-value expression e. In this case, the backward execution of the statement sets $live_f(v)$ to 0. Here as well, forward execution non-deterministically sets values to $live_f(v)$.

Section 5 includes experimental results for an implementation of both of the analyses described in this section.

4 Computing the Meet of Heap Abstractions

In this section, we develop a meet algorithm for a family of abstract domains and discuss the complexity of the algorithm for two cases: (i) for arbitrary 3-valued structures, and (ii) for bounded structures.

4.1 The Problem Setting

Our aim is to provide an algorithm applicable for a family of abstract domains based on 3-valued structures, including the abstract domain of bounded structures, $D_{\text{B-STRUCT}}$.

We design a meet algorithm for the domain $D_{\text{3-STRUCT}}$, which we consider as a basis for other abstract (sub-) domains. Given a sub-domain $\mathcal{D} \subseteq D_{\text{3-STRUCT}}$ and a set of abstract elements $X \in \mathcal{D}$, the result of the algorithm is possibly not an element of $\mathcal{D}$. However, when $\bigsqcap_{\mathcal{D}} X$ is defined, the inequality $\bigsqcap_{\mathcal{D}} X \sqsubseteq \bigsqcap_{D_{\text{3-STRUCT}}} X$ holds. Therefore, a domain specific operator $Refine_{\mathcal{D}} : D_{\text{3-STRUCT}} \to \mathcal{D}$ can be used to refine the result to yield an element of $\mathcal{D}$, $Refine_{\mathcal{D}}(\bigsqcap_{D_{\text{3-STRUCT}}} X) = \bigsqcap_{\mathcal{D}} X$.

For certain abstract domains, including $D_{\text{B-STRUCT}}$, no refinement is required. We now explain this formally.

Definition 6. *We say that an abstract domain $\mathcal{D} \subseteq D_{3\text{-}STRUCT}$, with the same ordering between abstract elements as in $D_{3\text{-}STRUCT}$ (see Definition 4), is* meet-admissible *when it satisfies the following conditions.*

Sublattice of $D_{\text{3-STRUCT}}$. *$\mathcal{D}$ is a lattice, and $\bigsqcap_{\mathcal{D}} X = \bigsqcap_{D_{3\text{-}STRUCT}} X$ and $\bigsqcup_{\mathcal{D}} X = \bigsqcup_{D_{3\text{-}STRUCT}} X$ for every finite subset X of $\mathcal{D}$.*

Closure of singletons. *For every structure $S \in 3\text{-}STRUCT$, if S exists in some set $XS \in \mathcal{D}$ then $\{S\} \in \mathcal{D}$. This condition allows us to break the problem of computing meet on sets of structures to a set of sub-problems where meet is computed on pairs of structures.*

Theorem 1. *The (parametric) abstract domain of bounded structures, $D_{B\text{-}STRUCT}$, is meet-admissible.*

The following proposition reduces the problem of computing the meet of two sets of structures to the problem of computing the meet of two structures by using the join operator, which we discuss at the end of this section.

Proposition 1. *Let XS_1, XS_2 be two elements in a meet-admissible domain $\mathcal{D}$. Then,*

$$XS_1 \sqcap XS_2 = \bigsqcup_{\substack{S_1 \in XS_1 \\ S_2 \in XS_2}} \{S_1\} \sqcap \{S_2\} \ . \tag{2}$$

In the remainder of this section, we consider the following problem. Given two structures $S_1, S_2 \in$ 3-STRUCT, compute $\{S_1\} \sqcap \{S_2\}$.

4.2 Computing the Meet of Two Structures

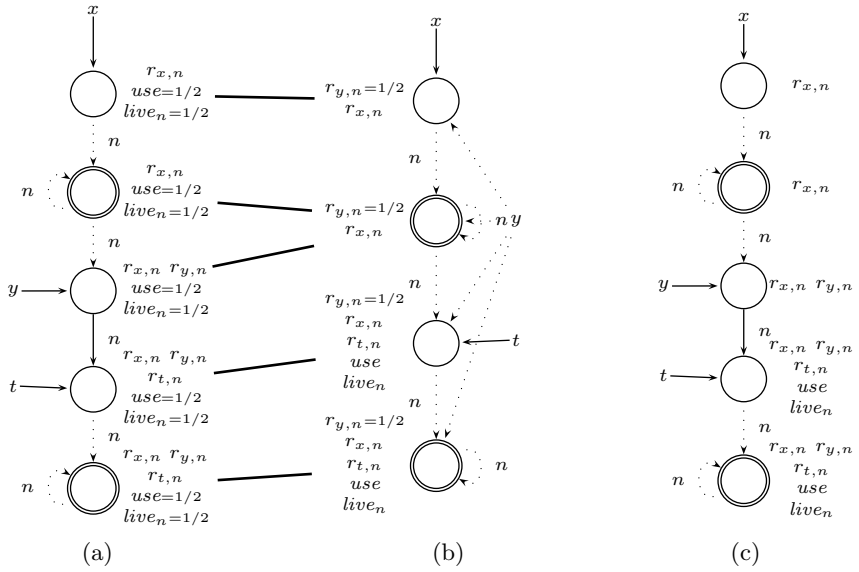

Fig. 4: An example for computing meet for the running example. (a) A structure that arises during the forward shape analysis; (b) A structure that arises during the backward (object liveness) analysis. (c) The meet of (a) and (b)

Fig. 4 shows two structures and their meet. (For now, ignore the edges between the structures in Fig. 4(a) and Fig. 4(b).) The structure in Fig. 4(a) arises during forward shape analysis, after the statement $t = y.n$ at line 10 of the running example; this is the structure from Fig. 2(b) with non-deterministic assignments to the values of the predicates $puse(v)$ and $live_n(v)$. The structure in

Fig. 4(b) is obtained from the structure in Fig. 3(b) by backward execution of the statement $y = t$ at line 11 of the running example. The meet of these two structures results in the structure shown in Fig. 4(c).

We now establish a connection between the structures that comprise the result of meet and certain relations that hold between their individuals. We first define the meet of two Kleene values t_1 and t_2. If $t_1 \sqsubseteq t_2$ then $t_1 \sqcap t_2 = t_1$, if $t_2 \sqsubseteq t_1$ then $t_1 \sqcap t_2 = t_2$, and otherwise the result is undefined and we denote it by the special symbol $\perp$.

Definition 7 (Meet Correspondence). *Given two structures $S_1 = (U_1, \iota_1)$ and $S_2 = (U_2, \iota_2)$, a relation $M \subseteq U_1 \times U_2$ is a* meet correspondence *between S_1 and S_2 when it is: (a)* Full*, i.e.,*

$$\forall u_1 \in U_1 : \exists v_2 \in U_2 : u_1 \, M \, v_2 \text{ and } \forall v_2 \in U_2 : \exists u_1 \in U_1 : u_1 \, M \, v_2 \ ;$$

and (b) Consistent*, i.e., for every predicate p of arity k, and a pair of k-tuples $u_1, \ldots, u_k \in {U_1}^k$ and $v_1, \ldots, v_k \in {U_2}^k$, such that $u_i \, M \, v_i$ for $i = 1 \ldots k$, $p^{S_1}(u_1, \ldots, u_k) \sqcap p^{S_2}(v_1, \ldots, v_k) \neq \perp$.*

The structures in Fig. 4(a) and Fig. 4(b) have exactly one meet correspondence, which is shown by the edges between their individuals.

We can use a meet correspondence to construct a common lower bound of two structures in the following way.

Definition 8. *Given a meet correspondence M between structures $S_1 = (U_1, \iota_1)$ and $S_2 = (U_2, \iota_2)$, the operation $S_1 \sqcap_M S_2$ yields the M-*induced *structure $S = (U, \iota)$, where $U = \{\langle u, v\rangle \in M\}$, and the interpretation of every predicate p of arity k and every k-tuple of nodes $\langle u_1, v_1\rangle, \ldots, \langle u_k, v_k\rangle \in U^k$ is given by*

$$p^S(\langle u_1, v_1\rangle, \ldots, \langle u_k, v_k\rangle) = p^{S_1}(u_1, \ldots, u_k) \sqcap p^{S_2}(v_1, \ldots, v_k) \ .$$

We are now ready to characterize the result of the meet operator in terms of meet correspondences.

Theorem 2. *Let $\mathcal{M}_{S_1,S_2} \subseteq \wp(U_1 \times U_2)$ denote the set of meet correspondences between structures S_1 and S_2. Then,*

$$\{S_1\} \sqcap \{S_2\} = \bigsqcup_{M \in \mathcal{M}_{S_1,S_2}} \{S_1 \sqcap_M S_2\} \ .$$

Theorem 2 already gives us a naive way to compute meet by: (a) Enumerating all relations $M \in U_1 \times U_2$; (b) Checking each of them to see whether it constitutes a meet correspondence; (c) For each meet correspondence, computing $S_1 \sqcap_M S_2$, and (d) Combining the results via join. Although the meet of two structures is a set of structures containing $2^{|U_1 \times U_2|}$ structures in the worst case, the size of the set is usually small, in practice. Notice that the above approach is intractable even when the number of structures is small, since the majority of the relations are not meet correspondences.

An immediate consequence of [10] is that deciding whether the meet of two arbitrary 3-valued structures is empty is NP-complete. The next theorem states that meet is computationally hard even for two bounded structures.

Theorem 3. *Given two bounded structures S_1 and S_2, the problem of deciding whether $\{S_1\} \sqcap \{S_2\} \neq \emptyset$ is NP-complete.*

Since the problem of computing meet with polynomial worst-case complexity is hard, we aim to achieve good efficiency in practice. We develop an algorithm based on a strategy. The strategy exploits certain properties of the abstract domain to prune the set of relations and find the meet correspondences. In Section 5, we supply empirical evidence showing that the algorithm successfully prunes most irrelevant relations when used in an abstract interpreter for inferring temporal heap properties on several benchmark programs.

4.3 Enumerating Meet Correspondences

We now present a strategy for exploring the (exponential) space of relations between two structures, searching for meet correspondences. The strategy, shown in pseudo-code in [1], attempts to prune relations that do not constitute a meet correspondence as much as possible, and relies on another procedure for solving a graph-matching problem on graphs (explained below).

The strategy consists of 4 stages that are run consecutively:

1. **Consistency of nullary predicates.** If there exists a nullary predicate p such that $p^{S_1}() = 1$ and $p^{S_2}() = 0$ or $p^{S_1}() = 0$ and $p^{S_2}() = 1$, then the result of meet is the empty set.
2. **Removing infeasible node pairs.** We remove from the set $U_1 \times U_2$ node pairs $\langle u, v \rangle$ such that there exists a predicate p of arity k and $p^{S_1}(u^k) \sqcap p^{S_2}(v^k) = \bot$, where u^k denotes a k-tuple containing the node u in all k positions. By Definition 7 these pairs are not contained in any meet correspondence.
3. **Finding full relations.** To satisfy the fullness requirement of Definition 7, we solve the following graph matching problem. Given a graph $G = \langle V, E \rangle$ and a subset W of V, find all subsets $M \subseteq E$ such that in the graph $\langle V, M \rangle$ the degree of every vertex is at least 1, and for vertices in W the degree is at most 1. In our case, $V = U_1 \cup U_2$, E is the set of pairs from the previous stage, and W is the set of non-summary nodes. An (worst-case exponential time) algorithm for this problem that uses several heuristics to solve this problem efficiently is discussed in [1].
4. **Consistency test.** The full relations from the previous stage are tested for consistency according to Definition 7 (in polynomial time). The relations that pass the test are meet correspondences and are used to create M-induced structures which are combined via join to yield the result.

The intuition behind the second stage is that two structures, possible produced by different analyses, may share a common set of unary predicates that are assigned only definite values, i.e., 0 or 1. Usually, these are the predicates that represent reference variables. In such cases, these predicates help prune many of the infeasible edges and determine a subset of edges with degree 1, which

must participate in every meet correspondence. Our algorithm uses these edges to reduce the amount of searching that has to be done.

In Fig. 4, the first stage of the algorithm is degenerate, as there are no nullary predicates. The second stage prunes the set of all node pairs, which consists of 20 pairs, to 5. This reduction occurs since the predicates x, t, $r_{x,n}$, and $r_{t,n}$ have definite values in both structures. In this example, there is only one full relation, which is returned by the third stage of the algorithm. This relation is indeed consistent, and thus the structure in the output is produced.

4.4 Computing Join

The join of sets of 3-valued structures is set union, followed by removal of non-maximal structures. To remove non-maximal structures, we need an algorithm to check for whether a structure $S_1 = (U_1, \iota_1)$ is embedded in a structure $S_2 = (U_2, \iota_2)$.

We observe that an embedding relation (see Definition 3) is actually a meet correspondence that satisfies a stricter version of the consistency condition. It is: (i) Full; and (ii) for every predicate p of arity k, and a pair of k-tuples $u_1, \ldots, u_k \in {U_1}^k$ and $v_1, \ldots, v_k \in {U_2}^k$, such that $u_i \, M \, v_i$ for $i = 1 \ldots k$, $p^{S_1}(u_1, \ldots, u_k) \sqsubseteq p^{S_2}(v_1, \ldots, v_k)$.

Since checking the second condition for two structures can be done in polynomial time, we can reuse the techniques for finding meet correspondences. In the first stage we check that for every nullary predicate p, $p^{S_1}() \sqsubseteq p^{S_2}()$. In the second state we remove from the set $U_1 \times U_2$ all node pairs $\langle u, v \rangle$ such that there exists a predicate p of arity k and $p^{S_1}(u^k) \not\sqsubseteq p^{S_2}(v^k)$. We then proceed by enumerating full relations over the remaining node pairs to find one that fulfills the second condition of the embedding relation.

For arbitrary 3-valued structures, checking embedding is NP-complete. However, for bounded structures our algorithm decides the problem in polynomial time. This is because, for two bounded structures, an embedding relation, if one exists, is unique and completely determined by the unary predicates.

5 Inferring Temporal Properties for Compile-time Memory Management

Compile-time GC is most desirable for lightweight Java-based platforms, such as JavaCard, where the penalty induced by a runtime GC is sometimes intolerable due to the limited space and processing power. Such platforms normally provide a mechanism for explicit memory deallocation, e.g., through a free directive.

We have implemented the phased bidirectional analysis described in Section 3 in the TVLA system to infer compile-time GC information. Our analysis infers information for producing a set of free statements that can be safely added to the program to free unused objects. Moreover, our analysis ensures that an object is deallocated at the earliest possible time, i.e., immediately after the object is last used.

5.1 Experimental Results

Table 2 shows our benchmark programs, which were used in [9].[6] The first four programs involve manipulations of singly-linked lists. `DLoop` and `DPairs` involve manipulations of doubly-linked lists. The `small-javac` example was used in [8], where it has been shown that a significant potential for compile-time GC exists by manually rewriting the code to include null assignments. Our assign-null analysis is able to yield the manual rewriting automatically.

On all benchmark programs, both our compile-time GC and assign-null analyses were able to detect all opportunities for object deallocation and safe assignment of null to reference fields, respectively. This information allows the reclamation of unused space at the earliest possible time. For example, considering the program in Fig. 1, the compile-time GC analysis was able to determine the safe deallocation of the object pointed by y right after line 10, thus deallocating list elements as soon as they are being traversed. Our assign-null analysis was able to verify that a `y.n=null` assignment could be inserted after line 10. The analyses proved similar properties for the other benchmark programs.

Table 2. Benchmarks and analysis costs (in seconds and Mb.)

Program	Description	Forward		Backward	
		Time	Space	Time	Space
`Loop`	Running example (Fig. 1)	0.9	1.0	1.6	1.8
`CReverse`	Constructive list reversal	3.0	2.0	5.7	4.2
`Delete`	Deletion of a list element	12.4	3.2	41.1	12.9
`DLoop`	Doubly-linked list variant of `Loop`	1.4	1.3	2.3	2.5
`DPairs`	Doubly-linked list traversal in pairs	3.0	2.0	5.5	4.0
`small-javac`	Emulation of JavaC's parser facility	528.9	32.1	334.4	77.6

Table 2 shows the costs of the analysis on the benchmark programs. As both analyses have very similar costs, we only show the results of the compile-time GC analysis.

The experiments were conducted on a 1.6 GHz laptop with 512 Mb. of memory, running Windows XP.

In addition to analysis time and space, we measured two redundancy factors related to our meet algorithm. We evaluated the efficiency of the graph matching algorithm in stage 3. The results show that for all benchmark programs, at most 0.5% of the expanded search space did not lead to valid matchings. We also measured the percentage of full relations computed during stage 3 of the algorithm that did not constitute meet correspondences (eliminated in stage 4). In all benchmarks the average number of relations that were eliminated did not exceed 0.3%, and in most benchmarks no eliminations occurred.

We believe that our meet algorithm is efficient for the the following reason. The forward shape analysis produces very precise information, which means that the values of the unary shape predicates (in Table 1) are almost always

[6] The programs are available from www.cs.tau.ac.il/∼rumster/ctgc_benchmarks.zip.

definite. When the backward phase computes the backward effect of a statement, it accepts a structure where all unary predicates are definite and assigns non-deterministic values to only a fixed number of unary predicates—y and $r_{n,y}$ in the structure shown in Fig. 4(b). Then, the meet is applied to structures where most unary shape predicates predicates have definite values. Our algorithm is geared to exploit these situations by focusing the search for meet correspondences.

6 Related Work

Computing Meet of Heap Abstractions. In [3], a meet is used for interprocedural shape analysis. Two algorithms are presented for computing meet on bounded structures. The first algorithm uses a "canonicalization"[7] operation to transform the structures to sets of structures in the image of canonical abstraction with the same concretization. Computing meet for the resulting structures is then straightforward. However, canonicalization can unnecessarily increase the number of structures by an exponential factor in the worst case. In our examples the worst case would indeed manifest itself, since we set the values of the temporal heap properties to non-deterministic values. Our algorithm avoids this problem by operating directly on the given structures. The second algorithm approximates meet by transforming one of the structures into a dynamic set of constraints and using a constraint solver. While usually more efficient than the first algorithm, it computes an over-approximation of meet. We believe that our algorithm can be used to improve the running times of the interprocedural analysis reported in [3].

In [4], it is shown how to compute meet for a class of formulas that precisely characterize bounded structures. The computation is essentially achieved in the same way as the first algorithm in [3].

In [11], a symbolic semi-algorithm for meet is presented. The algorithm converts bounded structures to formulas, and then uses logical conjunction to compute the result in the domain of formulas. Converting the resulting formula back to bounded structures is done via a theorem prover. The algorithm operates with respect to a finer concretization function than the one defined in Section 2. Specifically, this concretization function is parameterized by a set of integrity constraints C, and is defined by

$$\gamma_C(S) = \left\{ S' \in \text{2-STRUCT} \mid S' \sqsubseteq S,\, S' \models C \right\} \ .$$

The advantage of this algorithm is that it provides the most precise result with respect to γ_C. However, its performance can be quite low, due to the use of canonicalization and a potentially large number of calls to a theorem prover.

A distinct advantage of the algorithm presented in this paper is that it is not restricted to bounded structures and works for any set of 3-valued structures.

[7] Canonicalization is a semantic reduction akin to substituting abstract elements by their respective set of join-irreducibles.

Compile-time Memory Management. Most of the work on compile-time GC analysis has been done for functional languages. This paper demonstrates a compile-time GC analysis that applies to an imperative language with destructive updates, and is capable of reclaiming an object that is still reachable, but not used further in the run.

In [9], a user-specification-driven compile-time GC and assign-null analysis are described. The user specifies a *free query* of the form (pt, x), where pt is a program location and x is a program variable. A positive answer to the query means that a `free x` statement may be issued after program point pt. In contrast, the algorithms in this paper do not require nor rely on a user-specified queries, but rather perform an analysis on an exhaustive set of queries generated automatically using a simple heuristic. We believe our approach may be significantly more efficient compared to the analysis of [9] with our exhaustive set of queries.

References

1. G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Intersecting heap abstractions with applications to compile-time memory management. Technical Report TR-2005-04-135520, Tel Aviv University, Apr 2005. Available at http://www.cs.tau.ac.il/∼rumster/TR-2005-04-135520.pdf.
2. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.
3. B. Jeannet, Alexey L., T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Proc. Static Analysis Symp.* Springer, 2004.
4. V. Kuncak and M. Rinard. Boolean algebra of shape analysis constraints. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, pages 59–72. Springer, January 2004.
5. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proc. Static Analysis Symp.*, pages 280–301, 2000.
6. Z. Manna and A. Pnueli. A hierarchy of temporal properties (invited paper). In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 377–410, 1989.
7. 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.
8. R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 104–113. ACM Press, June 2001.
9. R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proc. of Static Analysis Symposium (SAS'03)*, volume 2694 of *LNCS*, pages 483–503. Springer, June 2003.
10. G. Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2003. http://www.cs.tau.ac.il/∼gretay/.
11. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference (TACAS 2004)*, pages 530–545. Springer, March 2004.