Intersecting Heap Abstractions with Applications to Compile-time Memory Management

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, namely the common value of a set of abstract memory stores. This problem proves to have many applications in shape analysis, such as interpreting program conditions, refining abstract configurations, reasoning about procedures, and proving temporal properties of heap-manipulating programs, either via greatest fixed point approximation over the trace semantics, or in a staged manner over the collecting semantics. However, computing the meet of heap abstractions is non-trivial; its definition as the least upper bound of all lower bounds does not lead to an effective algorithm. We describe a constructive formulation of meet that is based on finding certain relations between abstract heap objects. The enumeration of those relations is reduced to finding constrained matchings over bipartite graphs. A simple heuristic is applied in order to reduce computational overhead, and is supposed to behave well for common real-life scenarios. We describe a prototype implementation of the algorithm for proving temporal heap properties via staged analysis. It is applied to obtain information for compile-time garbage collection on several small but interesting Java programs.

1 Introduction

This paper addresses the problem of computing the intersection of dynamic memory 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 all operands. Specifically, for two dynamic memory (also known as *heap* and *store*) abstractions, the corresponding meet value stands for the set of common stores that are represented by its operands.

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 [9] 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.

1.1 The Usefulness of Meet for Shape Analysis

Common wisdom in program analysis is that an efficient *join* operator, used to merge information along different control flow paths, is normally sufficient for solving dataflow problems.[4] However, the ability to effectively compute the *meet* of abstract elements, namely the abstract value that represents the common configurations implied by them, is found to be useful in many circumstances. In particular, it is useful for a variety of problems concerning heap abstractions. For example, reasoning about temporal properties of heap-manipulating programs requires combining (past) shape information, which is naturally computable by forward analysis, and information about the future (e.g., reference liveness information), which is naturally computable by backward analysis. Such combinations can be naturally formulated via meet (the idea of combining forward and backward analyses using meet is heavily used elsewhere though, e.g., see [21, 27]). A particular instance of this approach is aimed at an automatic discovery of dead memory objects and reference fields, which can be used conduct static garbage collection and improve runtime GC performance.

Nonetheless, a meet operator proves to be useful in other cases as well. For example, it can be used to approximate the effect of code blocks, e.g., when applying interprocedural analysis [17]. Another interesting application of the meet is to refine an analysis according to a semantic condition. This is similar to the *focus* operation of TVLA [23]. In particular, it ensures the possibility of conducting strong destructive pointer updates. These and additional applications are described in Section 3.

1.2 Main Results

We describe a solution to the problem of computing the meet operator for heap abstractions. For generality, abstractions are defined using 3-valued logic, following [26], which defines a rich family of heap abstractions.

The main contributions of this paper are summarized as follows:

- We present an effective algorithm to compute the meet of sets of 3-valued structures. We explain how the meet operator can be computed by finding certain relations between abstract heap objects, and systematically develop an algorithm to enumerate these relations.
- We provide an exemplified survey of new applications of meet for proving program properties, specifically for checking temporal safety properties by combining forward and backward analyses, and refinement of abstract values according to predefined semantic criteria (see Section 3). In one case, the use of meet is shown to surpass current techniques used in TVLA (namely, the *focus* operation [26]).
- We have implemented the meet algorithm in TVLA—a system for generating program analysis from operational semantics [23]—and used it to implement

[4] Dually, [20, 19, 32] only require a meet operator.

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.

1.3 Running Example

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

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

Fig. 1 shows a simple program in a Java-like language that processes the elements of a singly-linked list. This program serves as the running example in the rest of the 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 in the text after that line), and (b) that the reference field `n` of the object pointed-to by `y` is never used after line 10. The second fact is more challenging to prove, as the object pointed-to by `y` is different on every iteration of the loop.

1.4 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. Section 3 motivates the need for using a meet operator, in addition to join, for various abstract interpretation problems, and in particular 3-valued logic based analyses. In Section 4, we present a new algorithm for meet. Section 5 describes a new program analysis for obtaining compile-time garbage collection information in

Java programs that uses meet to combine information from a backward analysis with information from a forward analysis. Section 6 discusses related work. Formal proofs appear in the respective appendices.

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 [26].

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, $\iota(p) : U^k \to \{0, 1\}$.*

Throughout the rest of this paper we assume that the set of predicates includes the binary predicate *eq*, and insist that it is interpreted as equality between individuals.

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 of pointer variables in a program is denoted by $PVar$

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?

Table 1 shows the predicates used to record properties of individuals for the analysis of our running example. We also define additional so-called "instrumentation" predicates to capture properties of individuals such pointer-aliasing, sharing, cyclicity, and transitive reachability. As observed in [26], instrumentation predicates provide for more precise information when applying abstraction on a concrete semantics. In particular, in Table 1 we define instrumentation predicate that capture reachability information (via predicates of the form $r_{x,n}(v)$), sharing information (via the predicate $is(v)$) and information on cycles in the heap graph (via the predicate $c_n(v)$).

In this paper, program states (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 an individual u, appears next to the corresponding node. If a unary predicate represents a reference variable, then it is shown by having an arrow drawn from its name to the node pointed-to 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. We make an exception for *eq*, which is not drawn since any two nodes are different.

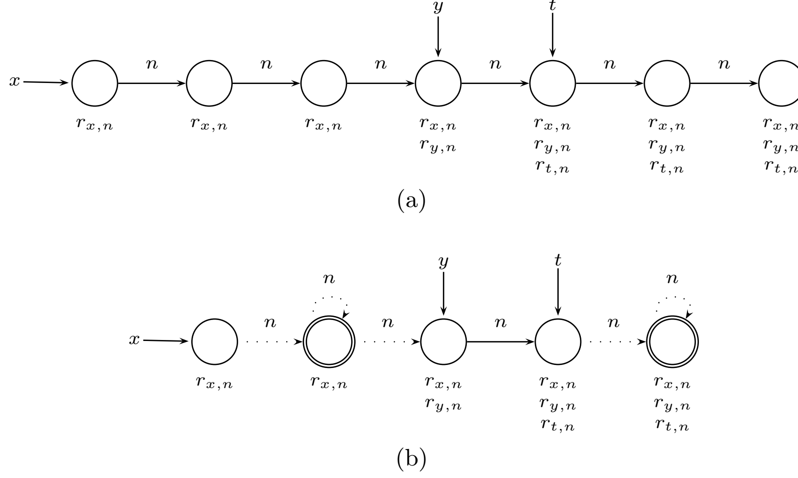

Fig. 2: (a) A concrete program state arising after the execution of the statement $t = y.n$; (b) An abstract program state approximating the concrete state in (a)

Fig. 2(a) shows a concrete program state arising after the execution of the statement $t = y.n$ on line 25 of the running example in Fig. 1. This state consists of a 7-elements singly-linked list, where x points to the first element of the list (i.e., the predicate $x(v)$ holds for the first element as shown by the edge connecting x and the first element), and the variables y and t point to the fourth and the fifth element of the list, respectively. In addition, all list elements are reachable from x through of an n path (i.e., the predicate $r_{x,n}(v)$ holds for all the nodes in this state). Finally, the fourth element of the list is reachable from y through an n path (i.e., $r_{y,n}(v)$ holds for this element), and the fifth through seventh elements of the list are reachable from both y and t (i.e., both $r_{y,n}(v)$ and $r_{t,n}(v)$ hold for these elements).

2.2 Abstract Program States

We consider a family of abstract domains sharing common properties. First, we describe the representation of abstract program states and the ordering relation

between abstract elements. Next, we present a *core* abstract domain that subsumes all of the abstract domains belonging to the family. Finally, we 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 abstract program states we use are based on 3-valued logic [26], 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.*

In this paper, 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.

We denote the set of all 3-valued logical structures over a set of predicates $\mathcal{P}$ by 3-STRUCT[$\mathcal{P}$], usually abbreviating 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* $u_1, \ldots, u_k \in U$*,*

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

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 can be 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* $XS_1, XS_2 \in$ *3-STRUCT,* $XS_1 \sqsubseteq XS_2$ *if and only if* $\forall S_1 \in XS_1 : \exists S_2 \in XS_2 : S_1 \sqsubseteq S_2$.

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 $D_{3\text{-}STRUCT}$ consists of all finite sets of* 3*-valued structures that do not contain non-maximal structures,* $\{XS \subset 3\text{-}STRUCT \mid \forall S_1, S_2 \in XS : S_1 \sqsubseteq S_2 \implies S_1 = S_2\}$*, partially ordered as in Definition 4*[5]*.*

Together with the obvious definitions for the bottom element (empty set), top element (two structures, one with an empty universe and the other containing a single summary node, where all predicates are interpreted as $1/2$ *for every node-tuple assignment), join operator (set union minus non-maximal structures), and meet operator (defined via join),* $D_{3\text{-}STRUCT}$ *forms a lattice.*

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 [26], 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. This function is parameterized by a special set of predicates A, referred to as *abstraction predicates*.

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

The abstraction function $\beta^{\mathcal{P},A}_{blur}$: 2-STRUCT$[\mathcal{P}] \rightarrow$ 3-STRUCT$[\mathcal{P}, A]$ converts a (potentially unbounded) 2-valued structure into an A-bounded 3-valued structure, by merging all A-equivalent individuals, i.e., individuals with the same values for all predicates in A. Namely, $\beta^{\mathcal{P},A}_{blur}((U, \iota)) = (U', \iota')$, where U' is the set of A equivalence classes of U, and the interpretation ι' of each predicate $p \in \mathcal{P}^{(k)}$ and each 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 an A-bounded structure obtained from the structure in Fig. 2(a), with $A = \mathcal{P}^{(1)}$ (the set of all unary predicates).

The abstraction function β_{blur} serves as the basis for abstract interpretation in TVLA [23]. 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. We also define the function α, which extends β_{blur} for sets of structures by $\alpha(XS) = \bigsqcup\{\beta_{blur}(S) \mid S \in XS\}$.

---

[5] Disallowing non-maximal structures ensures a partial order on the sets.

3 The Meet Operator and its Uses in Program Analysis

This section motivates the need for using meet operators—in addition to join—for program analysis. Most of the material in this section is well known and applicable to arbitrary lattices and Galois Connections. We demonstrate it using 3-valued structures to motivate the use of our algorithm.

3.1 Partial Interpretation of Program Conditions

The simplest application of meet operators is to partially interpret program conditions. In some cases, this enables to drastically improve the precision of program analysis by avoiding some infeasible control flow paths. The abstract effect of a program condition can be conservatively defined by

$$XS_{in} \sqcap XS_{cond} \ ,$$

where XS_{in} is a set of 3-valued structures representing the concrete states that may occur before the program condition, and XS_{cond} is a set of 3-valued structures that represents the program condition. In particular, the result is $\perp$ (the empty set) when the condition is not feasible, thus allowing the analysis to omit XS_{in} from the abstract values after the condition. This also allows the analysis to prove the absence of errors specified by certain conditions, e.g., cleanness conditions. When $XS_{in} \sqcap XS_{cond} \neq \perp$, a potential error is flagged.

The soundness of partial interpretation is immediate from the Galois connection $\wp(\text{2-STRUCT}) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} D_{\text{B-STRUCT}}$. In particular, let $[\![cond]\!] \sqsubseteq$ 2-STRUCT be the states for which a program condition *cond* holds. Then, for every $XS_{\text{in}} \in D_{\text{B-STRUCT}}$, the following equations hold:

$$\gamma(XS_{in}) \cap [\![cond]\!] \subseteq \gamma(XS_{in} \sqcap \alpha([\![cond]\!])) \tag{2}$$

$$\alpha(\gamma(XS_{in}) \cap [\![cond]\!]) \sqsubseteq XS_{in} \sqcap \alpha([\![cond]\!]) \ . \tag{3}$$

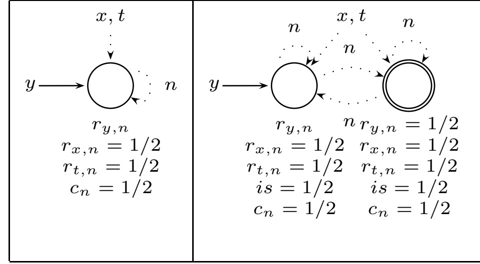

Fig. 3: 3-valued structures representing a program condition, or an abstraction refinement, where `y != null`

Example 1. Fig. 3 shows the 3-valued structure $XS_{\texttt{y != null}}$, which represents the program condition `y != null` at line 23 in the running example of Fig. 1. The partial interpretation of the program condition `y != null`, when the input structure XS_{in} is the 3-valued structure shown in Fig. 2(b), is obtained by $XS_{out} = XS_{in} \sqcap XS_{\texttt{y != null}}$, which yields, as expected, $XS_{out} = XS_{in}$. This is due to the fact that the `n` field of the object referenced by `y` in XS_{in} is not `null`.

The advantage of using meet is that it provides an effective way to approximate program conditions. Moreover, in many cases $\alpha([\![cond]\!])$ can be easily computed for certain forms of program conditions. For example, it is straightforward to define 3-valued structures that correspond to pointer equalities.

3.2 Refining 3-Valued Structures Based on Semantic Conditions

A meet operator can be used to refine a given abstract value, based on some semantic condition that does not directly correspond to the program syntax. For a given fixed set of 3-valued structures $\widehat{XS}$, the operation $\widetilde{\text{Focus}}_{\widehat{XS}} : D_{\text{3-STRUCT}} \to D_{\text{3-STRUCT}}$ is defined by

$$\widetilde{\text{Focus}}_{\widehat{XS}}(XS) = XS \sqcap \widehat{XS} \ .$$

Here, the structures of $\widehat{XS}$ are used to *refine* the abstract values of XS.

Example 2. An important issue in pointer analysis is handling destructive pointer updates. In order to guarantee that the statement `y.n = x` in line 5 of Fig. 1 is interpreted as a strong update, we may set $\widehat{XS}$ to be the set shown in Fig. 3, thus requiring that `y` points to a definite value. Notice that this idea is similar to program conditions. In fact, `y.n = x` is implemented by requiring that `y` is not `null`.

The refinement operation $\widetilde{\text{Focus}}$ is similar to the Focus operation implemented in TVLA [23], conforming to the specification of [26]. The TVLA operation refines the abstract value according to a first-order logical formula. Since first order formulas are more expressive than 3-valued abstractions, the TVLA focus is more expressible than the one obtained by a meet operator, and its user interface is more high-level. However, TVLA's Focus is incomplete in the sense that it is not well defined for every 3-valued structure and input formula. This is in line with the fact that the Focus operation generalizes the problem of first order satisfiability, which is undecidable.

In [4], it is shown that a meet-based focus operation can be used even in cases where TVLA's Focus is undefined (yielding an exception). The semantic condition used there requires that a singly-linked list has a last element, leading to an infinite number of structures in [26], however, resulting in a finite set of refined structured when meet is applied.

3.3 Backward Demand Shape Analysis

Demand shape analysis aims at proving that certain store properties cannot hold at a particular program point. For example, it is useful for verifying safety properties of stores, e.g., proving that a `null` dereference cannot occur at some program point, for *any* input.

While such properties can usually be revealed using ordinary (forward) analysis, previous work [15, 10] has shown that demand-driven (backward) analysis reduces the cost of an exhaustive analysis by answering a dataflow query, thus potentially requiring only partial expansion of the involved abstract domain configurations. It is commonly assumed that backward demand analysis can be directly derived from the forward exhaustive analysis: this can be achieved by reversing the effect of the forward collecting semantics underlying the abstract interpretation analysis—applying non-deterministic update for variables whose values are changed through some program statement—and interpreting the program statements counter flow-wise. Thus, in order to verify that some program configuration XS cannot occur at program location pt, it is sufficient to apply the resulted backward analysis on XS starting at pt, and verify that it yields an infeasible path, i.e., no valid configuration is associated with the beginning of the program.

Nonetheless, applying such techniques to (forward-based) shape analysis [26] yields an imprecise backward demand analysis. Primarily, this is due to the fact that the precision of shape analysis leans on past-related properties—such as sharing and reachability—whereas those are inaccessible to a backward analysis. In order to improve the precision of backward shape analysis, the collecting semantics can enforce further *feasibility constraints*—such that are derived from the program semantics—on stores associated with certain program locations. For example, in order for a program configuration XS to be feasible after a program statement of the form `lhs = rhs`, we require that, given XS, both `lhs` and `rhs` evaluate to the same value. This way, irrelevant configurations that might have occurred due to non-deterministic updating of variable values performed by the backward analysis, may be filtered out instead of being further propagated.

Clearly, feasibility conditions can be expressed in the form of first-order logical formulas. Nonetheless, similar to what was explained in Section 3.1 and Section 3.2, regarding the use of a meet operator for implementing abstraction refinement, such a technique can be naturally applied to the backward case as well: given some fixed set of 3-valued structures $\widehat{XS}_{st}$ representing the set of feasible configurations after a program statement st, the set of feasible program stores succeeding st can be conservatively obtained by

$$XS_{in} \sqcap \widehat{XS}_{st} \ .$$

Here, XS_{in} is a set of 3-valued structures representing the concrete states immediately succeeding st, yielded so far by the analysis.

Section 5 demonstrates backward demand shape analysis using meet-based feasibility conditions, as it was applied to verify safety properties of simple heap manipulating programs.

3.4 Interprocedural Analysis using Procedure-Specific Abstractions

In [17], the meet operator is used to conduct functional interprocedural analysis (see [31]). Here, we only sketch the main ideas.

Recall that the main problem in the functional approach to interprocedural analysis (and in structural dataflow analysis in general, e.g., [32]) is operating on representations of sets of transitions between concrete states. The effect of a code block B is a binary relation $\tau \subseteq$ 2-STRUCT $\times$ 2-STRUCT,

$$\tau = \left\{ (S_{in}, S_{out}) \mid B, S_{in} \rightsquigarrow S_{out} \right\} \ ,$$

where $B, S_{in} \rightsquigarrow S_{out}$ denotes the fact that the execution of B on S_{in} may terminate and yield a state S_{out}. Let τ_1 and τ_2 be relations on concrete states. The composition of τ_1 and τ_2 can be defined as

$$\left\{ (S_1, S_3) \,\middle|\, (S_1, S_2, S_3) \in \begin{array}{l} \left\{ (S_1, S_2, S') \mid (S_1, S_2) \in \tau_1,\, S' \in \text{2-STRUCT} \right\} \\ \quad \cap \left\{ (S'', S_2, S_3) \mid (S_2, S_3) \in \tau_2,\, S'' \in \text{2-STRUCT} \right\} \end{array} \right\} \ .$$

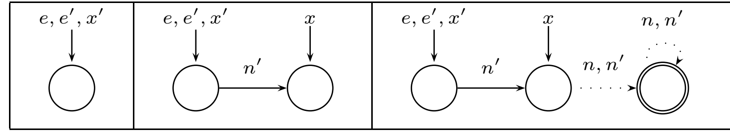

Fig. 4: A set of dual-vocabulary, 3-valued structures representing the effect of `x = prepend(e,x)`. Tagged predicates denote a posteriori properties

Every pair of concrete structures can be represented as a dual-vocabulary structure with two sets of predicates $\mathcal{P}_{in}$ and $\mathcal{P}_{out}$, representing S_{in} and S_{out}, respectively. This allows the relation τ to be conservatively represented using a set of dual-vocabulary, 3-valued structure with predicates for the values before and after a transition. For example, Fig. 4 shows a dual-vocabulary 3-valued structure set representing the effect of prepending an element pointed by `e` to a linked list pointed by `x`, before and after a call `x = prepend(e,x)`.

Since meet operations safely approximate intersections of concrete states, the composition of two dual-vocabulary, 3-valued structures $S_1^\sharp$ and $S_2^\sharp$ can be computed by

$$\left(S_1^\sharp \left[\mathcal{P}_{tmp} \leftarrow \mathcal{P}_{out}, \mathcal{P}_{out} \leftarrow 1/2 \right] \sqcap S_2^\sharp \left[\mathcal{P}_{tmp} \leftarrow \mathcal{P}_{in}, \mathcal{P}_{in} \leftarrow 1/2 \right] \right) \left[\mathcal{P}_{tmp} \leftarrow 1/2 \right] \ .$$

Here, an auxiliary temporary set of predicates $\mathcal{P}_{tmp}$ is used to match the output of $S_1^\sharp$ with the input of $S_2^\sharp$, thus employing a triple-vocabulary structure to simulate the composition. The operation $S[\mathcal{P}_{set} \leftarrow 1/2]$, for $set \in \{in, out, tmp\}$, sets the predicates of $\mathcal{P}_{set}$ of the triple-vocabulary structure S to $1/2$.

This technique, equipped with some further adjustments aimed to handle exchange of arguments and return values, is proved useful for interprocedural analysis by composing the effect of a procedure call—in the form of a dual-vocabulary 3-valued structure set XS_f—on some given program configuration XS_{in} that holds prior to that call. Furthermore, the use of a meet operator naturally provides for a rather modular approach, in the sense that neither parties—the caller nor the callee—needs a concrete notion of locally scoped properties of the other party (e.g., local variables). Hence, setting these properties to 1/2 provides for an immediate and effective approximation.

3.5 Verification of Temporal Properties via Trace Abstractions

Proving general temporal properties is challenging since some properties are only violated on infinite traces. In [8, Theorem 13] it is shown how to employ abstract interpretation to an upper approximation of the (infinite) set of possible (infinite) traces. In [34], an abstract interpretation algorithm for computing such an approximation was given. It represents traces using 3-valued structures. To guarantee soundness, the algorithm starts with $\top$ and computes greatest fixed points. On every iteration, a longer prefix of the trace is explored. As a result, the set of represented traces is reduced, until a fixed point occurs. The use of a meet operator allows to naturally implement such an iterative procedure by merging the longer traces with existing results. For more details refer to [8, Theorem 13].

3.6 Bidirectional Staged Verification of Temporal Properties

Certain temporal properties can be efficiently verified without explicitly representing traces. For example, a reference variable or object field is *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[6].

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

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

Here, it is assumed that the concrete 2-valued structures also record information on holding temporal properties. 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

[6] This is somewhat similar to *persistent* [24] properties that continuously hold from a given point in the trace.

is, their values upon termination of the execution). We write $(l_1, l_2), S_{in} \rightsquigarrow S_{out}$ 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 non-deterministically sets values of temporal properties. We write $(l_1, l_2), S_{out} \leftsquigarrow S_{in}$ 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 ($\sqcap$).

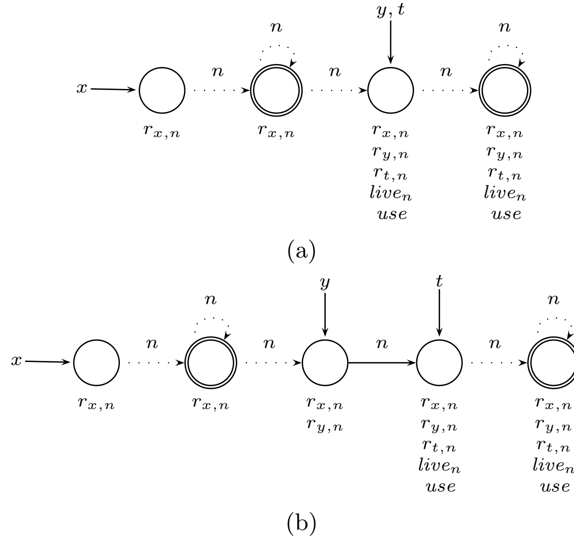

Fig. 5: 3-valued structures representing sets of program configurations, including heap object and reference field liveness, that arise (a) before the execution of the statement `t = y.n`; and (b) after it is executed

Example 3. We show how compile-time garbage collection information can be computed using a bidirectional staged 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. Technically, our analysis maintains a $use(v)$ predicate 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. Thus, in such a case, the backward execution of the statement $(l_1, l_2), S_{out} \leftrightarrows S_{in}$ 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. 5(a) shows one of the structures that arise before the statement `t = y.n` at line 25 of Fig. 1, and Fig. 5(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`. Verifying 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 free 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.

Example 4. Another application of bidirectional staged 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 an object reference field becomes *dead*, and therefore may be assigned a `null` value, thus significantly reducing potential GC drag time [28]. 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.

A reference field f of an object v is denoted *used* in a statement or a condition at edge (l_1, l_2), if a reference expression e—which is not an l-value—refers to the value of f. In this case, the backward execution of the statement $(l_1, l_2), S_{out} \leftrightarrows S_{in}$ 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 $live_f(v)$.

Fig. 5(a) shows one of the structures arising before the statement `t = y.n` at line 25 of Fig. 1, and Fig. 5(b) shows one of the structures arising after that statement. The `n` field of the object referenced by `y` is used at that statement, as it is reflected in $live_n(v)$ which holds for the individual referenced by `y`. However, that field is not being used any further prior to being redefined after the

statement, as $live_n(v)$ does not hold for the individual referenced by `y`. Verifying that $live_n(v)$ does not hold for any individual v referenced by `y`, for all structures arising after the statement, it follows that a `y.n = null` statement may be inserted after `t = y.n`, thus dropping the redundant reference and allowing the runtime GC to reclaim the space of the object held by `y.n` in a timely manner. Here as well, since the `n` field of the object pointed by `y` is live *for all structures* arising before the statement `t = y.n`, setting it to `null` right after that statement releases the reference as soon as possible.

Section 5 demonstrates bidirectional staged analysis as it was applied to conservatively approximate the liveness of heap objects and reference fields, thus implementing a proof of concept for the above examples.

4 Computing the Meet of Heap Abstractions

In this section, we develop a meet algorithm for a family of abstract domains.

4.1 Problem Setting

Our aim is to provide a single algorithm applicable for a wide family of abstract domains for shape analysis, and in particular for the abstract domain of bounded program states, $D_{\text{B-STRUCT}}$, presented in the previous section. We design an algorithm for a very general abstract domain $D_{\text{3-STRUCT}}$, which we consider as a basis for other abstract (sub-) domains. However, given an abstract domain $\mathcal{D} \subseteq D_{\text{3-STRUCT}}$ and abstract elements $X \in \mathcal{D}$, the result of $\bigsqcap_{\mathcal{D}} X$ in $\mathcal{D}$ (when it exists) is possibly different from the result in $D_{\text{3-STRUCT}}$. In fact, when the greatest lower bound exists, the inequality $\bigsqcap_{\mathcal{D}} X \sqsubseteq \bigsqcap_{D_{\text{3-STRUCT}}} X$ holds. To rectify this situation, two solutions are possible: (a) Designing a domain-specific operator that takes the result of the algorithm ($\bigsqcap_{D_{\text{3-STRUCT}}} X$) and refines it to an element of $\mathcal{D}$ ($\bigsqcap_{\mathcal{D}} X$); and (b) Specifying a set of conditions that an abstract domain should to satisfy for the result to be correct.

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

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

Closure of singletons *For every structure $S \in$ 3-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.*

Computable join *The result of $\bigsqcup X$ is computable for every finite subset $X \in \mathcal{D}$. This condition allows combining the results of the algorithm on individual sub-problems to yield the final result.*

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

Proof. See Appendix A.3.

Since a meet operator is associative, and in any program analysis we consider the input is a finite number of sets (of 3-valued structures), it suffices to consider the problem of computing meet for just two sets. The following lemma reduces the problem of computing the meet of two sets to the problem of computing the meet of two structures.

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

$$XS_1 \sqcap XS_2 = \bigsqcup_{\substack{S_1 \in XS_1 \\ S_2 \in XS_2}} \{S_1\} \sqcap \{S_2\} \ . \tag{4}$$

Proof. See Appendix A.1.

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

4.2 Computing the Meet of Two Structures

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 $\bot$.

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.,*

$$\begin{aligned} u_1 \in U_1 &\implies \exists v_2 \in U_2 : u_1 \, M \, v_2 \\ v_2 \in U_2 &\implies \exists u_1 \in U_1 : u_1 \, M \, v_2 \ ; \end{aligned}$$

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 $i = 1 \ldots k$,*

$$p^{S_1}(u_1, \ldots, u_k) \sqcap p^{S_2}(v_1, \ldots, v_k) \neq \bot \ .$$

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,*

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

Proof. See Appendix A.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 whether it constitutes a meet correspondence; and (c) For each meet correspondence, compute $S_1 \sqcap_M S_2$ and combine the results via join. This straightforward approach is not tractable however, since it requires enumerating an exponential number of relations ($2^{|U_1| \times |U_2|}$), although many of those relations are possibly not meet correspondences. Unfortunately, as shown in Appendix A.5, meet is computationally hard for general 3-valued structures, and even for bounded structures. This is in line with (and an immediate consequence of) [35].

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 Fig. 6, attempts to prune relations that are not meet correspondences as much as possible, and relies on another procedure for solving a problem on bipartite graphs (explained below).

The strategy consists of 4 stages that are run in sequence:

1. **Checking consistency of nullary predicates.** A check is made to see whether the two input structures disagree on a nullary predicate (i.e., the interpretation of the predicate is 0 in one structure and 1 in the other). In such a case the result of meet is the empty set.
2. **Removing infeasible node pairs.** The initial set of possible node pairs, $U_1 \times U_2$, is pruned by removing 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. The consistency requirement in Definition 7 implies that these pairs are not contained in any meet correspondence.
3. **Finding full relations.** Recall the fullness requirement of Definition 7. In order to satisfy this requirement, we use an algorithm for solving the following graph matching problem.
   The input to the problem is a bipartite graph $\langle U, V, E \rangle$ and functions $Q_a : U \cup V \to \mathbb{Z}$ and $Q_b : U \cup V \to \mathbb{Z}$. The output consists of all subsets $M \subseteq E$ such that every node $u \in U \cup V$ is incident with at least $Q_a(u)$ edges, and

```
function MEET(S_1 = (U_1, ι_1), S_2 = (U_2, ι_2))
  /* Verify consistency of nullary predicates. */
  if exists p ∈ P^(0) such that p^{S_1}() ⊓ p^{S_2}() = ⊥ then return ∅
  /* Form candidate match edges by unary correspondence. */
  E := ∅
  foreach u ∈ U_1, v ∈ U_2 do
    if p^{S_1}(u^k) ⊓ p^{S_2}(v^k) ≠ ⊥ for all p ∈ P^(k), k > 0 then E := E ∪ {(u, v)}
  /* Set matching ranges based on eq. */
  Q_a := {w ↦ 1 | w ∈ U_1 ∪ U_2}
  Q_b := {w ↦ 1 | w ∈ U_1 ∪ U_2}
  foreach u ∈ U_1 do
    if eq^{S_1}(u, u) = 1/2 then Q_b := Q_b[u ↦ max{degree(u, E), 1}]
  foreach v ∈ U_2 do
    if eq^{S_2}(v, v) = 1/2 then Q_b := Q_b[v ↦ max{degree(v, E), 1}]
  /* Find full matchings and filter meet correspondences. */
  XS := ∅
  foreach M ∈ ENUMABMATCH(⟨U_1 ∪ U_2, E⟩, Q_a, Q_b) do
    if M ∈ M_{S_1,S_2} then XS := XS ⊔ {S_1 ⊓_M S_2}
  return XS
```

Fig. 6: An algorithm for computing the meet of structures S_1 and S_2 over a fixed a set of predicates $\mathcal{P}$. The notation u^k stands for a k-tuple containing the node u in all k positions

at most $Q_b(u)$ edges, in M. An (worst-case exponential time) algorithm for this problem is discussed in the next subsection.
We use the observation that a non-summary node u (i.e., $eq(u, u) = 1$) can be matched with exactly one node in a meet correspondence, but a summary node can be matched with potentially one or more nodes. This information, together with the pruned set of edges from the previous stage, is used to form matching ranges to create an input for the graph matching problem described above. The output is a set of full relations that are consistent with respect to some of the predicate values.

4. **Checking consistency of predicates with positive arities.** The algorithm accepts the relations from the previous stage and checks (in polynomial time) whether they satisfy the consistency requirement for the predicate values that were not checked in previous stages. These relations satisfy both requirements of a meet correspondence and are used to create M-induced structures that are then joined together to yield the final result.

4.4 Enumerating ab-Matchings in Bipartite Graphs

We consider the following graph matching problem.

Problem 1. Given an undirected graph $G = (V, E)$ and minimal and maximal ranges on nodes $Q_a, Q_b : V \to \mathbb{Z}$, such that $Q_a(u) \le Q_b(u)$ for all $u \in V$, find all $M \subseteq E$ such that $Q_a(u) \le \left|\{(u, v) \in M\}\right| \le Q_b(u)$ for all $u \in V$.

This problem generalizes the problem of enumerating b-matchings, by also assigning lower matching ranges to nodes.

Considering an input to Problem 1, we say that a node in the graph is *violated* if its lower range exceeds its degree, thus such range cannot be satisfied by any matching. We say that a node is *saturated* if its upper range is not greater than zero, thus none of its incident edges can be included in any matching. A node of zero degree is said to be *isolated*. We now consider a general strategy to solve the generalized matching enumeration problem.

Lemma 1. *Given an input to Problem 1, consider the following strategy:*

1. *If the graph is empty, return a set consisting of the empty matching.*
2. *Select some node $u \in V$.*
3. *If u is violated, return an empty set.*
4. *If u is either saturated or isolated, remove u and its incident edges from the graph, and return the solution to the new problem.*
5. *Select some edge $e \in E$ that is incident with u.*
6. *Return the union of the solutions of the following sub-problems:*
   (a) e-exclusive matchings: *remove e from the graph, and solve the new problem.*
   (b) e-inclusive matchings: *if e's other endpoint is not saturated, then remove e from the graph, decrement the lower and upper ranges of both endpoints by 1, solve the new problem, and add e to any matching in the returned solution.*

Applying this strategy yields a solution to the problem.

Proof. In Appendix A.4.

While an algorithm to enumerate generalized matchings can be immediately derived from the strategy described in Lemma 1, it does not necessarily expand the search space in an effective manner. In particular, the selection of nodes for expansion may significantly speed up the convergence of the recursion, as demonstrated by each of the following policies:

1. Determining terminal cases prunes entire sub-portions of the search space, hence violated nodes should be selected at highest priority.
2. Removing redundant nodes reduces the overhead induced by the presence of their incident edges, and so they should be selected and removed (along with their incident edges) at high priority.
3. For all other cases, nodes whose induced forking degree of the search space is minimal are considered better candidates for selection, as their selection at early stages suggests that the search process commits to mandatory edges at higher priority. This way, the potential volume of the expanded search suffix may be significantly reduced.

Our proposed solution associates an additional value with each node in the graph, and selects nodes with such minimal associated value at high priority.

Definition 9. *Given an input to Problem 1, we define the* residual combinatorial degree *of a node u to be*

$$\mathrm{RCD}(u, E, Q_a, Q_b) = \left|\left\{E' \subseteq \{(u,v) \in E\} \,\middle|\, Q_a(u) \leq |E'| \leq Q_b(u)\right\}\right| \ .$$

Put in another way, the RCD of a node, with respect to an edge set and lower and upper matching ranges, is the number of subsets of its incident edges set that satisfy its ranges. In the case of ordinary matching, where both ranges are 1 for every node, this value equals the node's degree. Clearly, only violated nodes have an RCD value of 0, and therefore will be selected at highest priority. Redundant nodes have an RCD value of 1, and so do nodes with a single incident edge and a lower range of 1—indicating a mandatory edge for any matching. For all other cases, this heuristic prefers nodes of lower degrees, but also nodes of lower ranges.

Fig. 7 shows the pseudo-code of the matching algorithm, which applies the minimal RCD heuristic on node selection of the strategy described in Lemma 1. The correctness of this algorithm follows from the underlying strategy.

5 Using Meet for Compile-time Memory Management

Following Section 3.6, the use of a meet operator for bidirectional staged analysis has been applied to conservatively approximate compile-time memory management related properties. The following sections describe two instances of this

```
function EnumAbMatch(⟨V, E⟩, Q_a, Q_b)
    /* If V is empty, return empty matching. */
    if V = ∅ then return {∅}
    /* Select some node of minimal RCD. */
    Let u be a node in min_{RCD(u,E,Q_a(u),Q_b(u))} V
    /* If node is violated, return no matching. */
    if degree(u, E) < Q_a(u) then return {}
    /* If node is isolated or saturated remove and recurse. */
    if degree(u, E) = 0 or Q_b(u) ≤ 0 then
        V := V ∖ {u}
        E := E ∖ {(u, v) ∈ E}
        return EnumAbMatch(⟨V, E⟩, Q_a, Q_b)
    /* Select some edge and remove it from the edge set. */
    Let (u, v) be an edge in E
    E := E ∖ {(u, v)}
    /* Recurse without selected edge. */
    𝓜 := EnumAbMatch(⟨V, E⟩, Q_a, Q_b)
    /* Decrement ranges, recurse, and add selected edge. */
    if Q_b(v) > 0 then
        Q_a := Q_a[u ↦ Q_a(u) − 1, v ↦ Q_a(v) − 1]
        Q_b := Q_b[u ↦ Q_b(u) − 1, v ↦ Q_b(v) − 1]
        𝓜' := EnumAbMatch(⟨V, E⟩, Q_a, Q_b)
        𝓜 := 𝓜 ∪ {M ∪ {(u, v)} | M ∈ 𝓜'};
    return 𝓜
```

Fig. 7: An algorithm for enumerating all matchings in a graph with lower and upper node matching ranges

approach, the former aimed at allowing compile-time garbage collection and the latter promoting earlier reclamation of unused space by a runtime garbage collector. Finally, a prototype implementation for static GC of Java programs is described, as well as actual experimental results for a set of small but interesting Java programs.

5.1 Free Analysis

The analysis described in this section aims at providing static garbage collection in Java programs, thus substituting a runtime GC mechanism. This feature is most desirable for lightweight Java-based platforms, such as JavaCard, where the penalty induced by runtime GC is sometimes intolerable due to limited space and processing power. Such platforms normally provide a mechanism for explicit memory deallocation, e.g., through a free directive.

The actual instantiation involves a bidirectional, dual-stage analysis to conclude *future usage* information for each heap allocated object, at all program locations, as it was sketched in Example 3. The first (forward) stage tracks shape related information (see Table 1) but keeps the $use(v)$ predicate value to be 1/2 for all individuals v, by that representing non-deterministic interpretation of it. The second (backward) stage assumes false (0) value for $use(v)$ for all v, then updates its value where a dereference expression evaluates to v. This stage updates shape (history) related predicates to 1/2 where they are affected by a backward execution of a statement (e.g., assignments), thus representing non-deterministic interpretation of those predicates.

An auxiliary phase, taking place at the end of the analysis, conservatively yields the actual locations where an object reference—in the form of a reference variable or a one-level field dereference—can be issued an explicit `free` directive. This is obtained by verifying that $use(v)$ does not hold for any individual v pointed by it, for all structures that may arise at that location. The safety of this reasoning follows from the soundness of the analysis described in Section 3.6.

5.2 Nullify Analysis

Another analysis concerning static reasoning on garbage collection is intended to assist a runtime garbage collector by dispensing object references that are *dead* at some program location, namely references that are not used prior to being re-assigned throughout any execution path starting that location. As garbage detection in Java is based on dynamic reachability analysis and has no notion of possible future usage, previous work [28] has shown that the reclamation of unused space might suffer considerable delays incurred by dead references. While static (reference) variable liveness analysis is a well-known technique in program analysis, we are interested in its extension for object reference fields.

Here again, bidirectional staged analysis is used to approximate *reference field liveness* information for each heap allocated object, at all program locations, as described in Example 4. Similar to the above (see Section 5.1), $live_f(v)$ predicate values are retained 1/2 (non-deterministic) during the forward stage

of the analysis, and are updated by the backward stage to true (1) or false (0) depending on the *use* or *definition* of reference expressions evaluating to the f field of individual v, respectively.

As in Section 5.1, an auxiliary phase is applied to conservatively yield the actual locations where, for some reference variable x and reference field f, an object reference field $x.f$ may be assigned a `null` value. This is achieved by verifying that $live_f(v)$ does not hold for any individual v for which x holds, for all structures that may arise at each of these locations. As in the previous case, the safety of the verification follows from the soundness of the analysis.

5.3 Experimental Results

Table 2. Description of the benchmark programs

Program	Description
`Loop`	Construct and traverse a linked list (Fig. 1)
`CReverse`	Constructive list reversal
`Delete`	Delete an element from a list
`DLoop`	Doubly-linked list variant of `Loop`
`DPairs`	Doubly-linked list traversal in pairs
`small-javac`	Emulation of JavaC's parser facility

Table 2 shows our benchmark programs. The first four programs involve manipulations of singly-linked lists. `DLoop` and `DPairs` involve manipulations of doubly-linked lists. `small-javac` is motivated by [28], where the the code of the JavaC compiler is manually rewritten, issuing null assignments to heap references. Our nullify analysis is able to yield the manual rewriting automatically.

For all benchmark programs, both our free and nullify analyses managed to determine exact object use and reference field liveness, respectively. Thus, they yielded accurate free and nullify information, respectively, such that allows the reclamation of unused space at the earliest possible time. For example, considering the program in Fig. 1, the free analysis was able to determine the safe deallocation of the object pointed by y right after line 25, thus deallocating list elements as soon as they are being traversed. Our nullify analysis was able to verify the safe `null` assignment to $y.n$ after line 25, which leads—together with the safe `null` assignment to x after line 22 (concluded easily by a traditional stack variable liveness analysis)—to earlier reclamation of further unused objects by the runtime garbage collector. In `CReverse`, the analyses show that the elements of a linked list can be deallocated or nullified as soon as they are copied to the reversed list. In `Delete`, it is shown that an object can be freed as soon as it is taken out of the list, although it is still reachable from a temporary variable. Similar properties are proved for the doubly-linked lists programs as well.

Table 3. Analysis costs, in terms of time, space, and meet redundancy measurements

Program	Forward		Backward		Meet redundancy %	
	Time	Space	Time	Space	Matching	Recursion
Free analysis						
`Loop`	0.89	0.96	1.60	1.80	0.0 (0.0)	0.7 (0.4)
`CReverse`	3.02	1.97	5.72	4.23	0.0 (0.0)	1.2 (0.3)
`Delete`	12.41	3.23	41.11	12.87	0.0 (0.0)	0.7 (0.03)
`DLoop`	1.39	1.29	2.30	2.45	0.0 (0.0)	1.7 (0.4)
`DPairs`	3.03	2.03	5.47	3.98	0.0 (0.0)	0.6 (0.2)
`small-javac`	528.89	32.08	334.41	77.57	8.1 (0.3)	3.1 (0.2)
Nullify analysis						
`Loop`	0.88	0.97	1.53	1.80	0.0 (0.0)	0.7 (0.4)
`CReverse`	3.00	1.97	5.68	3.94	0.0 (0.0)	1.4 (0.5)
`Delete`	12.47	3.11	43.47	13.90	0.0 (0.0)	0.8 (0.03)
`DLoop`	1.42	1.30	2.32	2.47	0.0 (0.0)	1.7 (0.4)
`DPairs`	3.05	1.84	6.07	4.23	0.0 (0.0)	0.5 (0.1)
`small-javac`	525.22	32.42	373.86	78.36	8.3 (0.3)	3.2 (0.2)

Table 3 shows the costs of the analyses as they were applied to the benchmark programs. Time is measured in seconds, and space is measured in MB. Meet redundnacy percentage applies to redundant enumerated matchings, as well as redundant recursion steps during the matching procedure, and indicates both overall and average case (in parentheses) redundancy. The measurement only considers the core analysis done by TVLA, as front-end (Java to TVLA) and back-end (extracting free and nullify information) computational overhead is insignificant, compared to that of TVLA. The experiments were done on a 1.6 GHz Pentium M machine with 512 MB of memory running Windows XP.

The above results give rise to our conjecture that both the construction of meet correspondences in a phased manner, and the heuristic-based approach for solving the sub-problem of bipartite graph matching, induce very low redundancy factors. In particular, they suggest that average case redundancy of both procedures of the meet algorithm do not exceed 0.5%, thus implying that the actual performance of the meet algorithm (per this benchmark set) is practically polynomial by the size of the output. In addition, our analyses generally perform within a factor of 2–4 times slower, compared to analyzing the same program for a single free or nullify query, as shown in [30].[7] This implies that our approach may be more profitable, as it discovers *all* possible satisfied queries, at a fixed cost factor and in an automatic manner. See Section 6.2 for further discussion.

[7] This is with the exception of the `small-javac` case, where the the forward-only analysis of [30] eliminates "garbage" object during the analysis, thus significantly reduces the abstract state space; unfortunately, such optimization could not be applied to our analysis, which involves a backward stage, due to technical shortcomings.

6 Related Work

6.1 Computing Meet of Heap Abstractions

In [17], a meet operator is used for interprocedural shape analysis (see Section 3.4). Two algorithms are presented for computing meet over the powerset domain of canonical abstraction.

The first algorithm describes a naive approach to obtain meet for canonically bounded structures, by first substituting them with their corresponding sets of *canonicalized* structures,[8] and then computing meet in pairs over the expanded sets. Computing meet for a pair of canonicalized structures takes polynomial time. However, canonicalization might induce an exponential increase in the size of the input sets. Contrary to that, our algorithm does not assume the input structures to be canonically bound, hence it corresponds to a meet operator for the general-case 3-valued structure powerset domain, thus providing better preciseness for problems involving structures that are (temporarily) not canonic (the staged bidirectional analysis in Section 3.6 is an actual example). Furthermore, as canonicalization is completely avoided, it keeps computational cost closer to the actual size of the output. A trivial example is applying meet to a pair of identical sets, containing a single canonically bounded structure: while the result is clearly the same set, and is formulated by our algorithm by directly revealing the single meet correspondence that holds, the first algorithm of [17] would expand a whole set of canonicalized structures, that collapse back to the same single structure. Note that, for a pair of properly canonicalized structures, the two approaches perform essentially the same.

The second proposed algorithm obtains a meet operator by transforming one of the operand structures into a dynamic set of constraints, then applying it to the other operand. While generally more efficient than the first algorithm, in order to retain feasibility the algorithm avoids the generation of certain constraints, implying that the yielded result is in fact an over-approximation of the actual meet value.

In [22], a class of formulas that precisely characterize canonically bounded structures, and form a Boolean algebra, is presented. Computing meet for a set of formulas of this class is done in an analogous way to the algorithm in [17]: first, a normalization step is applied to the formulas, similar to the canonicalization step in [17]; then, conjunction is used to compute the result.

In [36], a symbolic (semi-) algorithm for meet is presented. The algorithm converts canonicalized structures to formulas, then uses logical conjunction to compute the result in the domain of formulas. Converting the resulting formula back to the domain of structures is done with a theorem prover. Such a symbolic algorithm suggests that a finer concretization function than the one defined in Section 2 can be used. Specifically, this concretization function also accommodates a set of integrity constraints C, and is defined by

$$\gamma_C(S) = \left\{ S' \mid S' \sqsubseteq S,\, S' \models C \right\} \ .$$

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

The advantage of the symbolic algorithm lies within the fact that it provides the most precise result with respect to γ_C, and by that is apt to analyses that require ultra-high precision. However, its performance can be quite low, due to the use of canonicalization and a potentially large number of calls to a theorem prover.

6.2 Compile-time Memory Management

This section considers the free and nullify analyses, that were sketched in Section 3.6 and manifested in Section 5.

Our described free analysis falls in the category of compile-time garbage collection research, where static techniques are applied to identify and recycle garbage memory cells. Most of the work in this area has been concentrated on functional languages [5, 16, 11, 13, 18]. This paper demonstrates a free 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 recent work (e.g., [7]) escape analysis, which allows stack allocation of dynamic objects, has been applied to Java. This way, an object is deallocated as soon as its allocating method returns. While this technique has been proved useful, it is limited to objects that do not "escape" their allocating method. Contrary to that, our described technique applies to all program objects, and allows their deallocation before their allocating method returns.

In region-based memory management [6, 33, 2, 12], the lifetime of an object is predicted at compile-time. An object is associated with a memory region, and the allocation and deallocation of the memory region are inferred automatically during compile-time. An interesting future work would be instantiating our framework with a static analysis algorithm for inferring earlier deallocation of memory regions.

Liveness analysis [25] may be used in the context of a runtime GC to reduce the size of the root set (i.e., by ignoring dead stack or global variables) or to reduce the number of scanned references (i.e., ignoring dead heap references).

In [3, 1, 14], liveness information for root references is used to promote unused space reclamation. In [29], dynamic measurements are conducted to estimate the potential space savings gained by communicating the liveness of stack reference variables, global reference variables, and heap reference fields to a runtime garbage collector. The conclusion there is that heap liveness information incorporates a significantly larger potential for space savings than that associated with stack and global variables liveness information only. A straightforward way of communicating heap liveness information to a runtime GC is by assigning `null` to dead heap references. Such a technique is actually instantiated in this paper, using a staged static analysis algorithm.

In [30], a framework for verifying temporal heap safety properties is instantiated with static algorithms for compile-time memory management. These algorithms consist of a single forward phase, during which information regarding the history of execution is recorded by a heap safety automaton. The input to these algorithms consists of a user specification as for the temporal heap safety

property that is of interest, and the output is a conservative answer to whether or not the input property hold for for all program execution paths. For example, the free analysis of [30] takes as input a free property query of the form (pt, x), and returns as output a conservative answer as to whether or not `free x` can be safely inserted after program point *pt*. In contrast, the algorithms in this paper do not require nor rely on user-specified properties, but rather generate a set of valid properties automatically. Furthermore, since each analysis phase—forward and backward—is applied once to conservatively obtain all temporal properties that hold for all program locations, we believe that this approach may be significantly more efficient compared to a multiple query instantiation of the analysis in [30]. This conjecture is generally backed-up by the experimental results shown in Section 5.3.

7 Conclusion

In this paper we present a new algorithm for computing meet for a heap abstraction domain. We describe a set of heap analysis related problems that can be conservatively resolved given an effective meet operator. We present a new algorithm for computing meet, along with a heuristic approach that is aimed to enhance its performance for commonly encountered cases. The algorithm is implemented in the TVLA framework, and meet-based bidirectional staged analyses are instantiated to obtain compile-time garbage collection information. Experimental results show that such analyses yields precise results in some non-trivial cases, and that the meet algorithm performs with very low redundancy.

References

1. Ole Agesen, David Detlefs, and J. Eliot Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 269–279. ACM Press, June 1998.
2. Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 174–185. ACM Press, June 1995.
3. Andrew W. Appel. *Compiling with Continuations*, chapter 16, pages 205–214. CUP, 1992.
4. Gilad Arnold. Combining heap analyses by intersecting abstractions. Master's thesis, Tel Aviv University, October 2004.
5. Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, 1977.
6. Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Symp. on Princ. of Prog. Lang.*, pages 171–183. ACM Press, 1996.
7. Bruno Blanchet. Escape analysis for object oriented languages. Application to Java$^{\mathrm{TM}}$. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, pages 20–34. ACM Press, 1998.
8. Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comp. Sci.*, 277:47–103, April 2002.

9. Patrick Cousot and Radhia 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.
10. Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand driven interprocedural data flow analysis. In *Trans. on Prog. Lang. and Syst.*, 1998.
11. Ian Foster and William Winsborough. Copy avoidance through compile-time analysis and local reuse. In *Proceedings of International Logic Programming Sympsium*, pages 455–469. MIT Press, 1991.
12. Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 141–152. ACM Press, 2002.
13. G. W. Hamilton. Compile-time garbage collection for lazy functional languages. In *Memory Management, International Workshop IWMM 95*, volume 637 of *Lec. Notes in Comp. Sci.* Springer-Verlag, 1995.
14. Martin Hirzel, Amer Diwan, and Antony L. Hosking. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *Trans. on Prog. Lang. and Syst.*, 24(6):593–624, 2002.
15. Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, New York, NY, October 1995. ACM Press.
16. Katsuro Inoue, Hiroyuki Seki, and Hikaru Yagi. Analysis of functional programs to detect run-time garbage cells. *Trans. on Prog. Lang. and Syst.*, 10(4):555–578, October 1988.
17. Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. In *Proc. Static Analysis Symp.* Springer, 2004. To appear.
18. Richard Jones. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1999.
19. John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.
20. Gary A. Kildall. A unified approach to global program optimization. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 194–206, New York, NY, 1973. ACM Press.
21. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.
22. Viktor Kuncak and Martin 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.
23. Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *Proc. Static Analysis Symp.*, pages 280–301, 2000.
24. Zohar Manna and Amir 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.
25. Steven Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

26. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
27. Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Automatic removal of array memory leaks in Java. In *Int. Conf. on Comp. Construct.*, volume 1781 of *Lec. Notes in Comp. Sci.*, pages 50–66. Springer-Verlag, April 2000.
28. Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Heap profiling for space-efficient Java. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 104–113. ACM Press, June 2001.
29. Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Estimating the impact of heap liveness information on space consumption in Java. In *Int. Symp. on Memory Management*, pages 171–182. ACM Press, June 2002.
30. Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Mooly 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.
31. Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
32. Robert E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, 1981.
33. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Symp. on Princ. of Prog. Lang.*, pages 188–201. ACM Press, January 1994.
34. Eran Yahav, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying temporal heap properties specified via evolution logic. In *European Symposium on Programming Languages (ESOP)*, 2003.
35. Greta Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2003. http://www.cs.tau.ac.il/∼gretay/.
36. Greta Yorsh, Thomas Reps, and Mooly 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.

A Proofs

Recall the definition of the Hoare order for two sets X and Y:

$$X \sqsubseteq Y \iff \forall x \in X \ \exists y \in Y : x \sqsubseteq y \ . \tag{5}$$

Also, recall that $Z = X \sqcap Y$ if Z is a lower bound of X and Y if and only if

$$Z \sqsubseteq X \wedge Z \sqsubseteq Y \ , \tag{6}$$

and Z is greater than any lower bound of X and Y

$$Z' \sqsubseteq X \wedge Z' \sqsubseteq Y \implies Z' \sqsubseteq Z \ . \tag{7}$$

A.1 Proof of Proposition 1

Proof. The first two conditions over the abstract domain (complete meet- sub-semilattice of $D_{3\text{-STRUCT}}$ and closure of singletons) ensure that the right-hand side of the equation

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

is well-defined.

We show that the right-hand side of the equation is the meet of XS_1 and XS_2 by proving it satisfies Eq. (6) and Eq. (7).

From the definition of meet, for every $S_1 \in XS_1$ and $S_2 \in XS_2$, $\{S_1\} \sqcap \{S_2\} \sqsubseteq \{S_1\}$ and $\{S_1\} \sqcap \{S_2\} \sqsubseteq \{S_2\}$ hold. Since the elements in the abstract domain are partially ordered by the Hoare order, we have that $\{S_1\} \sqsubseteq XS_1$ and $\{S_2\} \sqsubseteq XS_2$, and from transitivity of partial orders it follows that $\{S_1\} \sqcap \{S_2\} \sqsubseteq XS_1$ and $\{S_1\} \sqcap \{S_2\} \sqsubseteq XS_1$. It is a known fact that for partial orders

$$XS_1 \sqcap XS_2 = \bigsqcup_{Z \in \mathcal{D} : Z \sqsubseteq XS_1 \wedge Z \sqsubseteq XS_2} Z \ .$$

Since the set of abstract elements of the form $\{S_1\} \sqcap \{S_2\}$ is a subset of the elements $\{Z \in \mathcal{D} : Z \sqsubseteq XS_1 \wedge Z \sqsubseteq XS_2\}$ and join is a monotone operator, Eq. (6) is satisfied.

Suppose XS' is a lower bound of XS_1 and XS_2. From the definition of Hoare order, we have that, for every structure $S' \in XS'$, exists structures $S_1 \in XS_1$ and $S_2 \in XS_2$ such that $S' \sqsubseteq S_1$ land $S' \sqsubseteq S_2$. As well, $\{S'\} \sqsubseteq \{S_1\}$ land $\{S'\} \sqsubseteq \{S_2\}$, and from the definition of meet $\{S'\} \sqsubseteq \{S_1\} \sqcap \{S_2\}$. Therefore, since join is monotone, we have that, for every structure $S' \in XS'$,

$$\{S'\} \sqsubseteq \bigsqcup_{\substack{S_1 \in XS_1 \\ S_2 \in XS_2}} \{S_1\} \sqcap \{S_2\} \ .$$

And again, since join is monotone, we have

$$XS' \sqsubseteq \bigsqcup_{\substack{S_1 \in XS_1 \\ S_2 \in XS_2}} \{S_1\} \sqcap \{S_2\} \ ,$$

and Eq. (7) is satisfied.

□

A.2 Proof of Theorem 2

We use the following lemma to establish that the right-hand side of the equation

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

is a lower bound of $\{S_1\} \sqcap \{S_2\}$.

Lemma 2. *Let M be a meet correspondence between two structures $S_1 = \langle U_1, \iota_1 \rangle$ and $S_2 = \langle U_2, \iota_2 \rangle$, and let $S = \langle U, \iota \rangle$ be the M-induced structure $S_1 \sqcap_M S_2$. Then, $S \sqsubseteq S_1$ and $S \sqsubseteq S_2$.*

Proof. To prove the lemma, we provide functions $f_1 : U \to U_1$ and $f_2 : U \to U_2$, and show that $S \sqsubseteq^{f_1} S_1$ and $S \sqsubseteq^{f_2} S_2$. Recall that every node in the universe of S corresponds to a pair of nodes $\langle u, v \rangle$ where $u \in U_1$ and $v \in U_2$. The functions f_1 and f_2 act as projections on the first and second element of a pair, respectively, $f_1(\langle u, v \rangle) = u$ and $f_2(\langle u, v \rangle) = v$.

From Definition 7, we know that M is a full relation. Therefore, f_1 and f_2 are total and surjective.

From the way that S is produced, we know that for every predicate p of arity k and k-tuple $\langle u_1, v_1 \rangle, \ldots, \langle u_k, v_k \rangle \in U^k$,

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

Therefore,

$$p^S(\langle u_1, v_1 \rangle, \ldots, \langle u_k, v_k \rangle) \sqsubseteq p^{S_1}(u_1, \ldots, u_k) = p^{S_1}(f_1(\langle u_1, v_1 \rangle), \ldots, f_1(\langle u_k, v_k \rangle))$$

and

$$p^S(\langle u_1, v_1 \rangle, \ldots, \langle u_k, v_k \rangle) \sqsubseteq p^{S_2}(v_1, \ldots, v_k) = p^{S_2}(f_2(\langle u_1, v_1 \rangle), \ldots, f_2(\langle u_k, v_k \rangle)) \ .$$

Therefore, f_1 and f_2 satisfy both conditions required from embedding functions, and $S \sqsubseteq^{f_1} S_1$ and $S \sqsubseteq^{f_2} S_2$. □

Corollary 1. *The abstract element $\bigsqcup_{M \in \mathcal{M}_{S_1,S_2}} \{S_1 \sqcap_M S_2\}$ is a lower bound of $\{S_1\} \sqcap \{S_2\}$.*

Proof. Follows from Lemma 2 and the fact that the join operator is monotone. □

The following lemma helps to establish that the right-hand side of the equation is greater than any lower bound of $\{S_1\} \sqcap \{S_2\}$.

Lemma 3. *Let S, S_1, and S_2 be structures such that $S \sqsubseteq S_1$ and $S \sqsubseteq S_1$. Then, there exists a meet correspondence M between S_1 and S_2, and an M-induced structure $S_M = S_1 \sqcap_M S_2$ such that $S \sqsubseteq S_M$.*

Proof. Since $S \sqsubseteq S_1$ and $S \sqsubseteq S_1$, there exist embedding functions f_1 and f_2 such that $S \sqsubseteq^{f_1} S_1$ and $S \sqsubseteq^{f_2} S_1$.

We now define a relation $M \subseteq U_1 \times U_2$ and show that it is a meet correspondence. The relation contains all pairs of nodes from U_1 and U_2 that are the target of the same node by the embedding functions:

$$M = \{\langle u, v\rangle \mid \exists a \in U \wedge f_1(a) = u \wedge f_2(a) = v\} \ .$$

Fullness of M. Since f_1 and f_2 are total, every node $u \in U_1$ is the target of some node $a \in S$ (i.e., $f_1(a) = u$). In addition, $f_2(a) = v$ for some node $v \in U_2$, and so $\langle u, v\rangle \in M$. Symmetrically, for every node $v \in U_2$ there exists a node $u \in U_1$ such that $\langle u, v\rangle \in M$. Therefore, M is a full relation.

Consistency of M. Let p be a predicate of arity k, and let $u_1, \ldots, u_k \in U_1^k$ and $v_1, \ldots, v_k \in U_2^k$ be k-tuples of nodes such that $u_i M v_i$ for $i = 1 \ldots k$. From the way M is defined, there exists a k-tuple of nodes $a_1, \ldots, a_k \in U^K$ such that $f_1(a_i) = u_i$ and $f_2(a_i) = v_i$ for $i = 1 \ldots k$. Since f_1 and f_2 are embedding functions, the inequalities $p^S(a_1, \ldots, a_k) \sqsubseteq p^{S_1}(u_1, \ldots, u_k)$ and $p^S(a_1, \ldots, a_k) \sqsubseteq p^{S_2}(v_1, \ldots, v_k)$ hold, and therefore $p^{S_1}(u_1, \ldots, u_k) \sqcap p^{S_2}(v_1, \ldots, v_k) \neq \bot$, which shows that M is consistent.

Since M is a meet correspondence, $S_M = S_1 \sqcap_M S_2$ is well defined, and what is left to show is that $S \sqsubseteq S_M$. We present a function $f : U \to U^{S_M}$, and show that $S \sqsubseteq^f S_M$. The function f is defined by $f(a) = \langle f_1(a), f_2(a)\rangle$.

Let $\langle u, v\rangle \in M$ be a node in the universe of S. From the construction of M, we know that there exists node $a \in U$ such that $f_1(a) = u$ and $f_2(a) = v$. Therefore, $f(a) = \langle f_1(a), f_2(a)\rangle$, which shows that f is surjective.

Now, let p be a predicate of arity k, and let $a_1, \ldots, a_k \in U^k$ be a k-tuple of nodes. Since $S \sqsubseteq^{f_1} S_1$ and $S \sqsubseteq^{f_2} S_2$, we know that $p^S(a_1, \ldots, a_k) \sqsubseteq p^{S_1}(f_1(a_1), \ldots, f_1(a_k))$ and $p^S(a_1, \ldots, a_k) \sqsubseteq p^{S_2}(f_2(a_1), \ldots, f_2(a_k))$. Therefore, $p^S(a_1, \ldots, a_k) \sqsubseteq p^{S_1}(f_1(a_1), \ldots, f_1(a_k)) \sqcap p^{S_2}(f_2(a_1), \ldots, f_2(a_k))$. From the definition of f and the definition of M, we have $f(a_i) = \langle u_i, v_i\rangle$ and $u_i M v_i$ for $i = 1 \ldots k$. Therefore, from the construction of S_M, we get

$$p^{S_M}(f(a_1), \ldots, f(a_k)) = p^{S_1}(f_1(a_1), \ldots, f_1(a_k)) \sqcap p^{S_2}(f_2(a_1), \ldots, f_2(a_k)) \ ,$$

and finally, we conclude that

$$p^S(a_1, \ldots, a_k) \sqsubseteq p^{S_M}(f(a_1), \ldots, f(a_k)) \ .$$

Therefore, $S \sqsubseteq^f S_M$.

□

Corollary 2. *The abstract element* $\bigsqcup_{M\in\mathcal{M}_{S_1,S_2}}\{S_1 \sqcap_M S_2\}$ *is greater than any lower bound of* $\{S_1\} \sqcap \{S_2\}$.

Proof. Follows from Lemma 3 and the definition of the partial order over abstract elements (Hoare order). □

We are now ready to complete the proof.

Proof (Theorem 2). From Corollary A.2 and Corollary 2 we have that $\bigsqcup_{M\in\mathcal{M}_{S_1,S_2}}\{S_1 \sqcap_M S_2\}$ is the meet of $\{S_1\}$ and $\{S_2\}$ in the domain $D_{\text{3-STRUCT}}$. We now use the assumption that the greatest lower bound of $\mathcal{D}$ is the same as that of $D_{\text{3-STRUCT}}$, and conclude that $\bigsqcup_{M\in\mathcal{M}_{S_1,S_2}}\{S_1 \sqcap_M S_2\}$ is also the greatest lower bound in $\mathcal{D}$. □

A.3 Proof of Theorem 1

Lemma 4. *The abstract domain* $D_{\text{B-STRUCT}}$ *is a sublattice of* $D_{\text{3-STRUCT}}$.

Proof. We have to show that $D_{\text{B-STRUCT}} \subset D_{\text{3-STRUCT}}$ is a lattice, and $\bigsqcap_{D_{\text{B-STRUCT}}} X = \bigsqcap_{D_{\text{3-STRUCT}}} X$ and $\bigsqcup_{D_{\text{B-STRUCT}}} X = \bigsqcup_{D_{\text{3-STRUCT}}} X$,for every subset X of $D_{\text{B-STRUCT}}$.

It is straightforward to verify that

$$D_{\text{B-STRUCT}} = \langle PB, \sqsubseteq, \bot_{D_{\text{B-STRUCT}}}, \top_{D_{\text{B-STRUCT}}}, \sqcup_{D_{\text{B-STRUCT}}}, \sqcap_{D_{\text{B-STRUCT}}} \rangle$$

is a lattice where:

- The set of abstract elements PB consists of all finite sets of bounded structures not containing non-maximal elements (with respect to the embedding relation);
- The order relation $\sqsubseteq$ is the same as the as order in $D_{\text{3-STRUCT}}$ (see Definition 4);
- The bottom element $\bot_{D_{\text{B-STRUCT}}}$ is the empty set (same as in $D_{\text{3-STRUCT}}$);
- The top element $\top_{D_{\text{B-STRUCT}}}$ is $\top_{D_{\text{B-STRUCT}}} = \{S_{\top 1}, S_{\top 2}\}$ where $S_{\top 1} = (\emptyset, \iota_\top)$ and $S_{\top 2} = (\{u\}, \iota_\top)$, and $\iota_\top(p)(v_1, \ldots, v_k) = 1/2$ for every predicate p of arity k and every node-tuple assignment (same as in $D_{\text{3-STRUCT}}$);
- The join operator $\sqcup_{D_{\text{B-STRUCT}}}$ is set union followed by removal of non-maximal structures (same as in $D_{\text{3-STRUCT}}$); and
- The meet operator is mathematically defined as $\bigsqcap X = \bigsqcup\{Y \mid Y \sqsubseteq X\}$.

The fact that $D_{\text{B-STRUCT}}$ and $D_{\text{3-STRUCT}}$ identify on their join is obvious (they use the same operator).

To show that $D_{\text{B-STRUCT}}$ is a sublattice of $D_{\text{3-STRUCT}}$, we rely on the fact that $\bigsqcap_{D_{\text{3-STRUCT}}}$ can be computed from Proposition 1 and Theorem 2, and show that if $X \subseteq D_{\text{B-STRUCT}}$ then $\bigsqcap_{D_{\text{3-STRUCT}}} X \in D_{\text{B-STRUCT}}$.

Notice that Proposition 1 and Theorem 2 ultimately apply (the same) join to structures computed by $S_1 \sqcap_M S_2$, where S_1 and S_2 belong to the input sets and M is a meet correspondence between S_1 and S_2 (the operator $\sqcap_M$ is defined in Definition 8). Since $D_{\text{B-STRUCT}}$ is closed under join, we only need to show

that if M is a meet correspondence between two bounded structures S_1 and S_2, then the structure $S_1 \sqcap_M S_2$ is also bounded.

Let M be a meet correspondence between two bounded structures S_1 and S_2, and let $S_M = (U^{S_M}, \iota^{S_M})$ be the structure $S_1 \sqcap_M S_2$. To show that S_M is bounded, we need to show that for every two distinct individuals $u, v \in U^{S_M}$ there exists a unary predicate p, such that either $p^{S_M}(u) = 1$ and $p^{S_M}(v) = 0$ or $p^{S_M}(u) = 0$ and $p^{S_M}(v) = 1$.

From the construction of S_M we know that $u = \langle a_1, b_1 \rangle$ and $v = \langle a_2, b_2 \rangle$, where $a_1, a_2 \in U^{S_1}$ and $b_1, b_2 \in U^{S_2}$. From the construction of S_M (every individual in U^{S_M} corresponds to exactly one pair in the relation M), we know that either $a_1 \neq a_2$ or $b_1 \neq b_2$. Without loss of generality, suppose that $a_1 \neq a_2$. Then, since S_1 is bounded, there exists a unary predicate p such that either $p^{S_1}(a_1) = 1$ and $p^{S_1}(a_2) = 0$ or $p^{S_1}(a_1) = 0$ and ${}^{S_1}p(a_2) = 1$. Using Lemma 2, we know that $p^{S_M}(a_1) \sqsubseteq p^{S_1}(a_1)$ and $p^{S_M}(a_2) \sqsubseteq p^{S_1}(a_2)$. Therefore, either $p^{S_M}(a_1) = 1$ and $p^{S_M}(a_2) = 0$ or $p^{S_M}(a_1) = 0$ and ${}^{S_M}p(a_2) = 1$. □

Proof (Theorem 1). Lemma 4 shows that $D_{\text{B-STRUCT}}$ satisfies the first condition from Definition 6 (sublattice of $D_{\text{3-STRUCT}}$). The second condition (closure of singletons) is also satisfied by the definition of $D_{\text{B-STRUCT}}$. The third condition (computable join) is satisfied, since the join operation consists of set union (computable in polynomial time) and removal of non-maximal structures (which could also be checked in polynomial time, since the structures are bounded). □

A.4 Proof of Lemma 1

Proof. We use structural induction on the reduction of the problem, to prove the correctness of the recursive approach.

Basis. Given an empty graph, the strategy yields the empty matching, which is the only valid matching. Hence, this is a correct solution to the case where $|V| = |E| = 0$.

Induction hypothesis. For some $n, m \geq 0$, such that either $n > 0$ or $m > 0$ (or both), assume that the strategy yields a correct solution to any input satisfying either $|V| < n$ and $|E| \leq m$, or $|V| \leq n$ and $|E| < m$.

Induction step. Let $G = (V, E)$ be a graph such that $|V| = n$ and $|E| = m$. We observe the different cases encountered.

- **Terminal cases.** Clearly, the discovery of a violated node implies that no matching can satisfy its lower quota, hence no valid matching exists. Returning the empty set, the strategy yields a correct solution.
- **Redundancy elimination.** As no edge that is incident with a saturated node can participate in a valid matching, the problem induced by the removal of such a node and its incident edges has a solution that is equivalent to the original problem. Similarly, an isolated node cannot affect any matching, thus is redundant to the solution. As these reduced problems have a smaller set of vertices, following the induction hypothesis the strategy yields the correct solution.

Partitioning. Obviously, for some selected edge e, the set of valid matchings can be partitioned to those including e and those excluding it. Then, the union of the solutions obtained for these two sub-problems yields a complete solution to the original one. We examine the recursive reduction applied by the strategy for both cases.

Exclusive partition. Consider the sub-problem induced by removing e from the edge set, and keeping all matching quotas intact. Clearly, any e-exclusive matching that satisfies the original problem, is also applicable to the new sub-problem, as e is not used and the quotas are the same. Obviously, any matching satisfying the new sub-problem is applicable to the original one. Following the induction hypothesis, the strategy yields the correct solution to this sub-problem.

Inclusive partition. If e's other endpoint is not saturated, consider the sub-problem induced by removing e from the edge set, and decrementing its endpoints' lower and upper quotas by 1. Clearly, any e-inclusive matching that satisfies the original problem, is applicable to the new sub-problem, having e removed from it, as the new quotas suffice for the rest of the edges that might be included. Also, any matching satisfying the new sub-problem is an applicable e-inclusive one to the original problem, having e added to it. Then, adding e to any solution obtained for the new problem, and following the induction hypothesis, the strategy yields the correct solution to this sub-problem.

Thus, we get that the union of these two partitions yields a correct solution to the original problem.

It follows that the strategy yields the correct solution to any valid input. □

A.5 The Copmutational Complexity of Meet

The next theorem addresses the complexity of the following decision problem related to the computation of meet.

The problem MEET accepts as input a pair of 3-valued structures, S_1 and S_2, and decides whether $S_1 \sqcap S_2 \neq \phi$.

Theorem 3. *MEET is NP-complete.*

Proof. We first show that MEET is in NP. Given structures $S_1 = \langle U_1, \iota_1 \rangle$ and $S_2 = \langle U_2, \iota_2 \rangle$, it is possible to non-deterministically choose a relation $M \subseteq U_1 \times U_2$ (polynomial in the size of the input), and check in polynomial time whether it is a meet correspondence.

To show that MEET is NP-hard we use a reduction from the problem 3-COLOR. The reduction takes as input an undirected graph $G = \langle V, E \rangle$ and returns two structures, $S3$ and $G3$, over the vocabulary $\{r^{(1)}, g^{(1)}, b^{(1)}, e^{(2)}\}$. The unary predicates $r(v)$, $g(v)$, $b(v)$ are used to represent color assignments to nodes (exactly one of the predicates holds for every node), and the binary predicate $e(u, v)$ represents graph edges. The structure $S3$ shown in Fig. 8 represents

all graphs with a consistent 3-coloring. The structure $G3 = \langle V, I3 \rangle$ represents the graph G with every possible assignments of colors to its nodes. This is achieved by interpreting the predicate $e(u, v)$ exactly as in G, and interpreting $I3(r)(v) = I3(g)(v) = I3(b)(v) = 1/2$ for all $v \in V$.

The result of meeting the two structures represents all possibilities of assigning a consistent 3-coloring to G. Therefore, $S3 \sqcap G3$ is non-empty if and only if G is 3-colorable. □

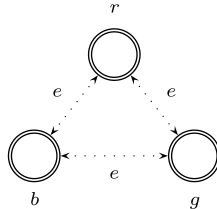

Fig. 8: Structure $S3$ that embeds all structures that correspond to graphs in 3-COLOR

The hardness result stated above also holds for a smaller class of inputs that consists of only bounded structures. Consider the problem B-MEET, which accepts as input a pair of bounded 3-valued structures, S_1 and S_2, and decides whether $S_1 \sqcap S_2 \neq \phi$.

Theorem 4. *B-MEET is NP-complete.*

Proof. The proof is a variation on the proof of the general case.

To show that B-MEET is in NP the same arguments as in the general case are used. To show that MEET is NP-hard, we again reduce from the problem 3-COLOR. Notice that structure the $G3$ used in the proof of the general case was already bounded. However, structure $S3$ was not bounded, which prevents the same proof from being used here. We supply a remedy for this situation by extending the vocabulary $\{r^{(1)}, g^{(1)}, b^{(1)}, e^{(2)}\}$ with a set of unary predicates $D = \{d_1^{(1)}, d_2^{(1)}, \ldots, d_k^{(1)} \mid k = \lceil \log |V| \rceil\}$, that are used to supply a unique canonical name to each node of $S3$. For every node v, the vector of values $(d_1(v), \ldots, d_k(v))$ is thought of as an id assigned to the node v.

The structure $S3$ interprets the predicates in D by assigning $1/2$ to every node and every predicate from D. Therefore, $S3$ represents all graphs with a consistent 3-coloring where the id of every node is any of the 2^k possibilities.

The structure $G3$ interprets the predicates in D such that each node receives a unique id vector. Therefore, $G3$ represents the graph G with unconstrained colors and an arbitrary id vector for each node.

Since the id vectors are unconstrained in $S3$, the result of meeting the two structures represents all possibilities of assigning a consistent 3-coloring to G (and assigning the same id vectors as those used in $G3$). Therefore, $S3 \sqcap G3$ is non-empty if and only if G is 3-colorable.

$\square$