Tel-Aviv University
The Raymond and Beverly Sackler
Faculty of Exact Sciences

Combining Heap Analyses by Intersecting Abstractions

Thesis submitted in partial fulfillment of the requirements for the
M.Sc. degree in Tel-Aviv University, The School of Computer Science

by

Gilad Arnold

Prepared under the supervision of Dr. Mooly Sagiv

September 2004

Acknowledgments

I would like to thank Mooly Sagiv, for the opportunity, the faith, and the invaluable guidance and support.

I owe a lot of credit to Ran Shaham, Eran Yahav, and a special huge gratitude to Roman Manevich, for their enormous efforts and continuous advising in bringing this work to a completion.

I thank G&D research program, and the Israeli Academy of Science, for their part in making this research possible.

Special thanks go to Zur Itzhakian and Ohad Barzilay, for the great time I had while working on this research.

Finally, to my beloved—Merav, Erel and Roie, and special thanks to Lea—this would not have happened without your consideration and support. I love you very much.

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 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 proposed algorithm for proving temporal heap properties via staged analysis. It is applied to obtain compile-time garbage collection on several small but interesting Java programs.

Contents

1 Introduction **4**

1.1 The Usefulness of Meet for Heap Analyses 4

1.2 Main Results 6

1.3 Running Example 7

1.4 Outline 9

2 3-Valued Shape Analysis Primer **10**

2.1 Concrete Program Configurations 10

2.2 Operational Semantics 13

2.3 Abstract Program Configurations 13

2.4 Bounded Program Configurations 15

2.5 Powerset Heap Abstraction 16

3 The Meet Operator and its Uses in Program Analysis **20**

3.1 Partial Interpretation of Program Conditions 20

3.2 Refining 3-Valued Structures Based on Semantic Conditions . 22

3.3 Backward Demand Shape Analysis 24

3.4 Interprocedural Analysis using Procedure Specific Abstractions 26

3.5 Verification of Temporal Properties via Trace Abstractions . . 29

3.6 Bidirectional Staged Verification of Temporal Properties . . . 29

4 Computing Meet for Sets of Heap Abstractions **35**

4.1 Computing Meet for a Pair of Abstract Heaps 36

4.2 Enumerating Meet Correspondences 39

4.3 Enumerating Full b-Matchings in a Bipartite Graph 40

4.4 Correctness 44

4.5 Performance 44

5 Applications and Experimental Results **47**

5.1 Free Analysis 47

5.2 Nullify Analysis 49

5.3 Prototype Implementation 50

5.4 Experimental Results 50

6 Related Work **55**

6.1 Computing Meet of Heap Abstractions 55

6.2 Enumerating Matchings in Graphs 57

6.3 Compile-Time Memory Management Analysis 58

7 Conclusion **61**

Bibliography **63**

A Proofs **70**

A.1 Proof of Lemma 4.1 70

A.2 Proof of Proposition 4.5 72

A.3 Proof of Lemma 4.6 73

A.4 Proof of Lemma 4.8 75

B Implementation **78**

B.1 Meet algorithm implementation 78

B.2 Reference set functionality 80

Chapter 1

Introduction

This thesis addresses the problem of computing the intersection of abstract dynamic memory configurations. 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, considering a pair of dynamic memory (also known as *heap*) configuration abstraction elements, the corresponding meet value stands for the set of common stores that are represented by its operands.

As it is undecidable to prove interesting properties even for reasonable programs, especially in the presence of dynamic memory allocation with pointers and destructive updates, the use of abstract interpretation [CC77] 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 Heap Analyses

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.[1] 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 information which is naturally computable by a forward analysis (e.g., shape information), and such that is naturally computable by a backward analysis (e.g., reference liveness information). Such combinations are naturally formulated with meet (the idea of combining forward and backward analyses using meet is heavily used elsewhere though, e.g., see [KRS94, SKS00]). A particular instance of this approach is aimed at an automatic discovery of dead memory objects and reference fields, such that provides for static garbage collection and improved runtime GC performance, respectively. An extension of previous work addressing this problem [SYKS03], this has been the main initiator for this thesis, and is further discussed in Section 3 and Section 5.

Nonetheless, a meet operator is proved useful for other cases as well. For example, it can be used to approximate the effect of code blocks, e.g., when applying interprocedural analysis [JLRS04]. 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 [LAS00]. In particular, it ensures the possibility of conducting strong destructive pointer updates. These and

[1] Dually, [Kil73, KU76, Tar81] only require a meet operator.

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 abstraction domains. For generality, abstractions are defined using 3-valued logic, following [SRW02], which defines a family of heap abstractions.

The main contributions of this thesis are summarized as follows:

- We present an effective algorithm for computing the meet operator over sets of 3-valued structures. It is derived from a new, alternative definition for formulating meet for heap abstraction domains, that is based on finding certain relations between abstract heap objects. The enumeration of these relations is reduced to solving a generalized matching problem over bipartite graphs. This induces a straightforward algorithmic method to obtain meet.

- In the general case, the result of a meet of arbitrary 3-valued structures might be exponential in the size of the input. Furthermore, even with respect to the size of the output, a polynomial bound cannot be proved, as this would reduce to solving NP-complete problems (this is an immediate consequence of [Yor03]). Nonetheless, in common real-life cases where operands of meet are proper bounded abstractions of their represented concrete counterparts, performance may increase significantly. Such effectiveness is achieved by applying a heuristic to reduce the expansion of the search involved with finding the formulating elements of the meet value. Our limited experience indicates that the algorithm is highly efficient in practice, normally performing with very low redundancy.

- An exemplified survey of new applications of meet for proving program properties, specifically such that address temporal safety properties by combining forward and backward analyses, and such that involve 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 [SRW02]), in the sense that—although somewhat more expressively restrictive—it is naturally defined and guaranteed to be computable.

- An implementation of the meet algorithm in TVLA—a system for generating program analysis from operational semantics [LAS00]: this, along with other implemented auxiliary mechanisms, allows TVLA users to employ analyses involving meet, in order to solve new heap concerned problems.

- An instantiation of a meet-based analysis, providing for a prototype for compile-time garbage collection in Java. This was used as a proof-of-concept for static verification of memory management properties, and was applied to several small yet interesting example programs. The results suggest that the analysis is sufficiently precise for capturing interesting dynamic store related properties, through static methods. Additionally, it was used for empirical evaluation of the proposed meet algorithm, which proved to induce fairly low redundancy by performing virtually polynomial by the size of the output.

1.3 Running Example

Fig. 1.1 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

```
1  class SLL {  /* Singly-linked list. */
2    public SLL n;
3    public int val;
4  };
5
6  class Main {
7    public static void main( String args[] ) {
8      SLL x, y, t;
9
10     /* List creation. */
11     x = null;
12     while (...) {
13       y = new SLL();
14       y.val = ...;
15       y.n = x;
16       x = y;
17     }
18
19     ...
20
21     /* List traversal. */
22     y = x;        /* x = null; */
23     while (y != null) {
24       System.out.print( y.val );
25       t = y.n;  /* free y;  or  y.n = null; */
26       y = t;
27     }
28   }
29 }
```

Figure 1.1: A program for creating and traversing a singly-linked list.

in this thesis. Instantiated compile-time garbage collection analyses, that rely on the use of a meet operator to intersect bidirectional analysis stages, are able to verify interesting properties: for example, the element pointed by y in line 25 can be statically deallocated right after t is assigned, thus assuring reclamation of unused space at the earliest possible time. Alternatively, it can be proved that **null** assignments to x right after line 22, and to y.n right after line 25, are safe, thus promoting earlier reclamation of space by a runtime GC.

1.4 Outline

The rest of the thesis 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 discusses experimental results concerning the actual implementation of staged bidirectional analysis and its application to obtain compile-time GC. Related work is discussed in Section 6. Section 7 gives concluding remarks. Finally, formal proofs, and a description of implementation related issues, appear in Appendix A and Appendix B, respectively.

Chapter 2

3-Valued Shape Analysis Primer

We give an overview of *first order transition systems* (FOTS), the formalism underlying the parametric analysis framework of [SRW02]. FOTS may be thought of as an imperative language built around an expression meta-language based on first-order logic with transitive closure.

2.1 Concrete Program Configurations

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

Definition 2.1. *A 2-valued logical structure over a set of predicates $\mathcal{P}$ is a pair $S^\natural = (U^\natural, I^\natural)$, where:*

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

Throughout the rest of this thesis we assume that the set of predicates

Core predicates	
$eq(v_1, v_2)$	Object v_1 equals object v_2
$ref(v)$	Reference variable ref points to object v
$fld(v_1, v_2)$	Field fld of object v_1 point to object v_2
Instrumentation predicates	
$r[ref, fld](v)$	Object v is heap-reachable from reference variable ref through a fld field path of objects
$is[fld](v)$	Object v is pointed by fld field of more than one object
$c[fld](v)$	Object v resides on a cycle along fld field path of objects

Table 2.1: Predicates used to represent shape information in the running example and their intended meaning.

includes the binary predicate eq, and insist that it is interpreted as equality between individuals.

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

In the following, we use $p^{I^\sharp}(v)$ as an alternative notation for $I^\sharp(p)(v)$, omitting the superscript $I^\sharp$ when no confusion is likely. The notation $\mathcal{P}^{(k)}$ refers to the set of predicates of arity k in $\mathcal{P}$. We denote the set of all 2-valued logical structures over a set of predicates $\mathcal{P}$ by $\mathcal{S}_2^{\mathcal{P}}$. We will mostly assume that the set of predicates $\mathcal{P}$ is fixed and abbreviate $\mathcal{S}_2^{\mathcal{P}}$ to $\mathcal{S}_2$.

Table 2.1 shows the predicates used to record properties of individuals for the analysis of our running example. A unary predicate $ref(v)$ holds when the reference (or pointer) variable ref points to the object v; in our example ref $\in \{\mathrm{x}, \mathrm{y}, \mathrm{t}\}$. Similarly, a binary predicate $fld(v_1, v_2)$ records the value of a reference (or pointer-valued) field fld; in our example fld $\in \{\mathrm{n}\}$.

We also define additional so-called "instrumentation" predicates to cap-

ture connectivity properties of individuals. As observed in [SRW02] instrumentation predicates provide for more precise information when applying abstraction on a concrete semantics. In particular, in Table 2.1 we define instrumentation predicate that capture reachability information (via the predicate $r[\mathit{ref}, \mathit{fld}](v)$), sharing information (via the predicate $\mathit{is}[\mathit{fld}](v)$) and information on cycles in the heap graph (via the predicate $c[\mathit{fld}](v)$).

In this thesis, program configurations (i.e., 2-valued logical structures) are depicted as directed graphs. Each individual of the universe is drawn as a node. The value of a nullary predicate $p()$ is denoted by writing its name and value in a box labeled "Nullary". A unary predicate $p(u)$, which holds for an individual u, appears next 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 by the variable. A binary predicate $p(u_1, u_2)$, 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 p. We make an exception for *eq*, which is not drawn since any two nodes are different.

Fig. 2.1(a) shows a concrete program configuration arising after the execution of the statement t = y.n at line 25 in the running example of Fig. 1.1. This configuration 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 field path of objects (i.e., the predicate $r[x, n](v)$ holds for all the nodes in this configuration). Finally, the fourth element of the list is reachable from y through a n field path (i.e., $r[y, n](v)$ holds for this element), and the fifth

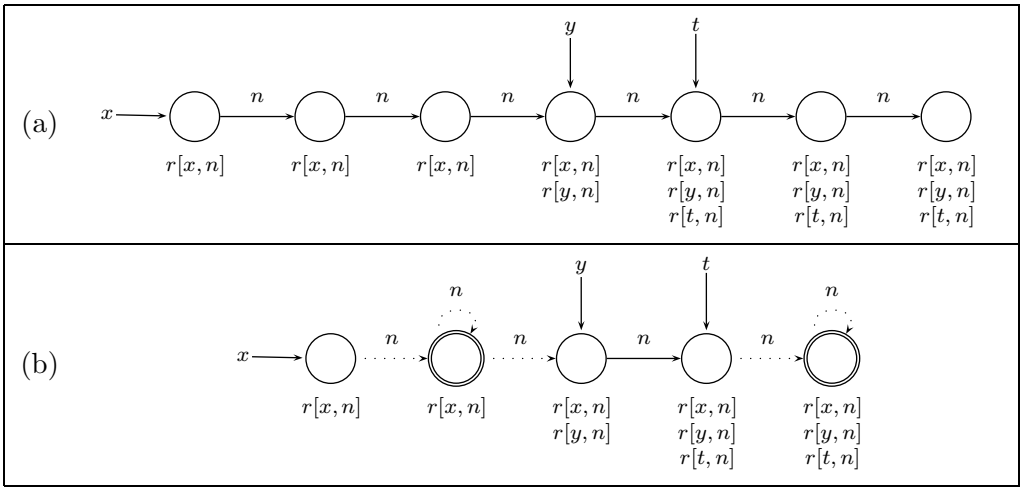

Figure 2.1: (a) A concrete program configuration arising after the execution of the statement t = y.n; (b) An abstract program configuration approximating the concrete configuration in (a).

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 Operational Semantics

In FOTS, program statements are modeled by *actions*, that specify how statements transform an inbound logical structure into an outbound logical structure. This is done primarily by defining the values of the predicates in the outbound structure using first-order logical formulae with transitive closure over the inbound structure [SRW02].

2.3 Abstract Program Configurations

We now describe the abstractions used to create a finite (bounded) representation of a potentially unbounded set of 2-valued structures (representing

heaps) of potentially unbounded size. The abstractions we use are based on Kleene 3-valued logic [SRW02], which extends Boolean logic by introducing a third value $\frac{1}{2}$, denoting values that may be either 0 or 1. In particular, we utilize the partially ordered set $\{0, 1, \frac{1}{2}\}$, where $0 \sqsubseteq \frac{1}{2}$ and $1 \sqsubseteq \frac{1}{2}$, with the join operation defined by

$$t_1 \sqcup t_2 = \begin{cases} t_1 & \text{if } t_1 = t_2, \\ \frac{1}{2} & \text{otherwise.} \end{cases}$$

Definition 2.2. *A 3-valued logical structure over a set of predicates $\mathcal{P}$ is a pair $S = (U, I)$, where:*

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

A 3-valued logical structure can be used as an abstraction of a larger 2-valued logical structure, that is, a 2-valued logical structure that contain more nodes. This is achieved by letting an abstract configuration (i.e., a 3-valued logical structure) contain *summary individuals*, namely, individuals that corresponds to one or more individuals in some concrete configuration represented by the abstract one. Formally, a summary individual u is such that $eq^I(u, u) = \frac{1}{2}$.

In this thesis, 3-valued logical structures are also depicted as directed graphs, where unary reference predicates, as well as binary, predicates with $\frac{1}{2}$ 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 $\mathcal{S}_3^{\mathcal{P}}$, usually abbreviating it to $\mathcal{S}_3$. We define a preorder on structures, denoted by $\sqsubseteq$, based on the concept of *embedding*.

Definition 2.3. *Let $S = (U, I)$ and $S' = (U', I')$ 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 $u_1, \ldots, u_k \in U$,*

$$p^I(u_1, \ldots, u_k) \sqsubseteq p^{I'}(f(u_1), \ldots, f(u_k)) \ . \tag{2.1}$$

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

Given a 3-valued structure S, $\gamma(S)$ denotes the set of concrete 2-valued structures that are embedded in S. We also extend γ for sets of 3-valued structures, point-wise. In this case, we say that $XS \in \wp(\mathcal{S}_3)$ *over-approximates* a set of concrete stores $XS' \in \wp(\mathcal{S}_2)$ if $XS' \subseteq \gamma(XS)$.

2.4 Bounded Program Configurations

Note that the size of a 3-valued structure is potentially unbounded and that $\mathcal{S}_3$ is infinite. The abstractions studied in this thesis 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 the *abstraction predicates*.

Let $A \subseteq \mathcal{P}^{(1)}$ be a set of unary predicates. An individual $u_1 \in U_1$ in a structure $S_1 = (U_1, I_1)$ is said to be A-*compatible* to an individual $u_2 \in U_2$ in a structure $S_2 = (U_2, I_2)$ iff, for every predicate $p \in A$, either $p^{I_1}(u_1) \sqsubseteq p^{I_2}(u_2)$ or $p^{I_2}(u_2) \sqsubseteq p^{I_1}(u_1)$.

A 3-valued structure is said to be A-*bounded* if no two distinct individuals in its universe are A-compatible. An A-bounded structure can have at most $2^{|A|}$ individuals. We denote the set of all A-bounded, 3-valued structures

over a set of predicates $\mathcal{P}$ by $\overline{\mathcal{S}}_3^{\mathcal{P},A} \subset \mathcal{S}_3^{\mathcal{P}}$, and allow to omit the superscripts when no confusion is likely.

The abstraction function $\beta_{blur}^{\mathcal{P},A} : \mathcal{S}_3^{\mathcal{P}} \rightarrow \overline{\mathcal{S}}_3^{\mathcal{P},A}$ converts a (potentially unbounded) 3-valued structure into an A-bounded, 3-valued structure, by merging all pairs of A-compatible individuals. Namely, $\beta_{blur}^{\mathcal{P},A}((U,I)) = (U',I')$, where U' is the set of A-compatible equivalence classes of U, and the interpretation I' of each predicate $p \in \mathcal{P}^{(k)}$ and each k individuals $c_1,\ldots,c_k \in U'$, is given by

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

Fig. 2.1(b) shows an A-bounded structure obtained from the structure in Fig. 2.1(a), with $A = \mathcal{P}^{(1)}$.

The abstraction function β_{blur} serves as the basis for abstract interpretation in TVLA [LAS00]. 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.

2.5 Powerset Heap Abstraction

This abstraction is based on the fact that there can only be a finite number of bounded structures that are *non-isomorphic* to one another. Note that two structures are isomorphic when there exists a bijection between their universes such that preserve the interpreted values of all predicate for all tuples of individuals. The powerset abstraction function bounds structures with respect to a set of abstraction predicates, and removes duplicate (isomorphic) structures.

For simplicity, we will consider *canonic* bounded structures. Note that the individuals of an A-bounded structure are uniquely identified by the set

of values induced by their interpretation to predicates in A. We refer to such a set of predicate values associated with an individual as the individual's *canonical name*. For example, the individual pointed by x in Fig. 2.1(b) has the canonical name $\{x \mapsto 1, y \mapsto 0, t \mapsto 0, r[x,n] \mapsto 1, r[y,n] \mapsto 0, r[t,n] \mapsto 0, c[n] \mapsto 0, is[n] \mapsto 0\}$. A canonic bounded structure is a bounded structure whose individuals are identified by their canonical names. We refer to the set of all canonic bounded structures by $\widehat{\mathcal{S}}_3^{\mathcal{P},A} \subset \overline{\mathcal{S}}_3^{\mathcal{P},A}$. Note that for some given $\mathcal{P}$ and A, $\widehat{\mathcal{S}}_3^{\mathcal{P},A}$ is finite.

Given some structure $S \in \mathcal{S}_3^{\mathcal{P}}$, the *canonic abstraction* function $\beta_{canonic}^{\mathcal{P},A} : \mathcal{S}_3^{\mathcal{P}} \to \widehat{\mathcal{S}}_3^{\mathcal{P},A}$ first applies $\beta_{blur}^{\mathcal{P},A}$ to S, then renames the individuals of the resulting structure by assigning them with their respective canonical names.

The powerset heap abstraction function $\alpha : \wp(\mathcal{S}_2) \to \wp(\widehat{\mathcal{S}}_3)$ is given by

$$\alpha(XS) = \{\beta_{canonic}(S) \,|\, S \in XS\} \ .$$

We use the Hoare ordering to obtain an order over sets of structures, as follows: for two sets of structures $XS_1, XS_2 \in \mathcal{S}_3$, we write $XS_1 \sqsubseteq XS_2$ iff

$$\forall S_1 \in XS_1 \ \exists S_2 \in XS_2 : S_1 \sqsubseteq S_2 \ ,$$

where $S_1 \sqsubseteq S_2$ is given by Definition 2.3. Note that Hoare ordering induces a *preorder*, as it may not satisfy antisymmetry.

In order to simplify the exposition of the material presented in this thesis, we refine the 3-valued structure powerset, thus making the Hoare ordering antisymmetric, by eliminating "redundant" structures. A set of 3-valued structure XS is *independent* w.r.t. embedding if, for all $S, S' \in XS$,

$$S \sqsubseteq S' \implies S = S'.$$

We denote by $\wp^{\not\sqsubseteq}(\mathcal{S}_3) \subset \wp(\mathcal{S}_3)$ the set of 3-valued structure independent sets. The operation $\Delta : \wp(\mathcal{S}_3) \to \wp^{\not\sqsubseteq}(\mathcal{S}_3)$ eliminates redundant structures, such

that $\Delta(XS) = XS'$ iff $XS' \subseteq XS$ and $XS \sqsubseteq XS'$. The following proposition summarizes the properties of 3-valued independent powerset abstraction domain.

Proposition 2.4. *Given the embedding partially ordered set* $(\mathcal{S}_3, \sqsubseteq)$, $\wp^{\sqsubseteq}(\mathcal{S}_3)$ *forms a complete lattice with respect to Hoare ordering, where:*

(a) $\bot = \emptyset$.

(b) $\top = \{(\emptyset, I), (\{u\}, I_{\frac{1}{2}})\}$, *where* $p^{I_{\frac{1}{2}}} u^k = \frac{1}{2}$ *for all* $p \in \mathcal{P}^{(k)}$.

(c) For some $P \subseteq \wp^{\sqsubseteq}(\mathcal{S}_3)$, *the join and meet operators are given by*

$$\bigsqcup P = \Delta(\bigcup P)$$
$$\bigsqcap P = \bigsqcup \left\{ XS' \in \wp^{\sqsubseteq}(\mathcal{S}_3) \,\middle|\, \forall XS \in P : XS' \sqsubseteq XS \right\} .$$

Proof. This is a well-known practice in lattice theory. □

In the following, we will use $XS_1 \sqcup XS_2$ and $XS_1 \sqcap XS_2$ as a shorthand for $\bigsqcup\{XS_1, XS_2\}$ and $\bigsqcap\{XS_1, XS_2\}$, respectively.

The following proposition establishes the technique for abstract interpretation used throughout this thesis.

Proposition 2.5. *Defining* $\alpha^{\sqsubseteq} : \wp(\mathcal{S}_2) \to \wp^{\sqsubseteq}(\widehat{\mathcal{S}}_3)$ *to be* $\alpha^{\sqsubseteq} = \Delta \circ \alpha$,

$$\wp(\mathcal{S}_2) \underset{\gamma}{\overset{\alpha^{\sqsubseteq}}{\rightleftarrows}} \wp^{\sqsubseteq}(\mathcal{S}_3)$$

forms a Galois connection.

Proof. This is an immediate extension to the Galois connection $\wp(\mathcal{S}_2) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \wp(\mathcal{S}_3)$, see [SRW02]. □

In the following, we will omit the explicit superscript notation and use α to denote the redundancy eliminating variant $\alpha^{\sqsubseteq}$.

From an algorithmic point of view, a join operator can be efficiently computed for canonic 3-valued structures, based on immediate matching of canonical names. However, the definition of the meet operator given in Proposition 2.4 is not tractable. It is given here as a strawman to demonstrate the algorithmic challenges involved with computing meet values. This definition exemplifies that every poset which is closed under join for all subsets, is also closed under meet.

Chapter 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 connection. We demonstrate it using 3-valued structures so as to motivate the use of the meet algorithm described in this thesis.

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

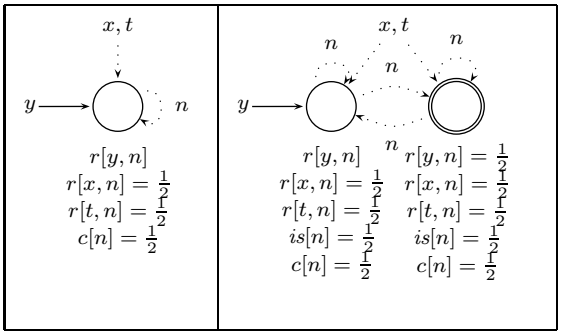

Figure 3.1: 3-valued structures representing a program condition, or an abstraction refinement, where y != **null**.

$\perp$ 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 $\wp(\mathcal{S}_2) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \wp^{\sqsubseteq}(\mathcal{S}_3)$. In particular, let $[\![cond]\!] \sqsubseteq \mathcal{S}_2$ be the states for which a program condition *cond* holds. Then, for every $XS_{\text{in}} \in \wp^{\sqsubseteq}(\mathcal{S}_3)$, the following equations hold:

$$\gamma(XS_{in}) \cap [\![cond]\!] \subseteq \gamma(XS_{in} \sqcap \alpha([\![cond]\!])) \tag{3.1}$$

$$\alpha(\gamma(XS_{in}) \cap [\![cond]\!]) \sqsubseteq XS_{in} \sqcap \alpha([\![cond]\!]) \ . \tag{3.2}$$

Example 3.1. Fig. 3.1 shows the 3-valued structure $XS_{\text{y != NULL}}$, which represents the program condition y != NULL at line 23 in the running example of Fig. 1.1. The partial interpretation of the program condition y != NULL, when the input structure XS_{in} is the 3-valued structure shown in Fig. 2.1(b), is obtained by $XS_{out} = XS_{in} \sqcap XS_{\text{y != NULL}}$, 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 which 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 which does not directly correspond to the program syntax. For a given fixed set of 3-valued structures $\widehat{XS}$, the operation $\widetilde{\text{Focus}}_{\widehat{XS}} : \wp^{\sqsubseteq}(\mathcal{S}_3) \to \wp^{\sqsubseteq}(\mathcal{S}_3)$ 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 3.2. An important issue in pointer analysis is handling destructive pointer updates. In order to guarantee that the statement y.n = x in line 15 of Fig. 1.1 is interpreted as a strong update, we may set $\widehat{XS}$ to be the set shown in Fig. 3.1, 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 [LAS00], conforming to the specification of [SRW02]. The TVLA operation refines the abstract value according to a first order

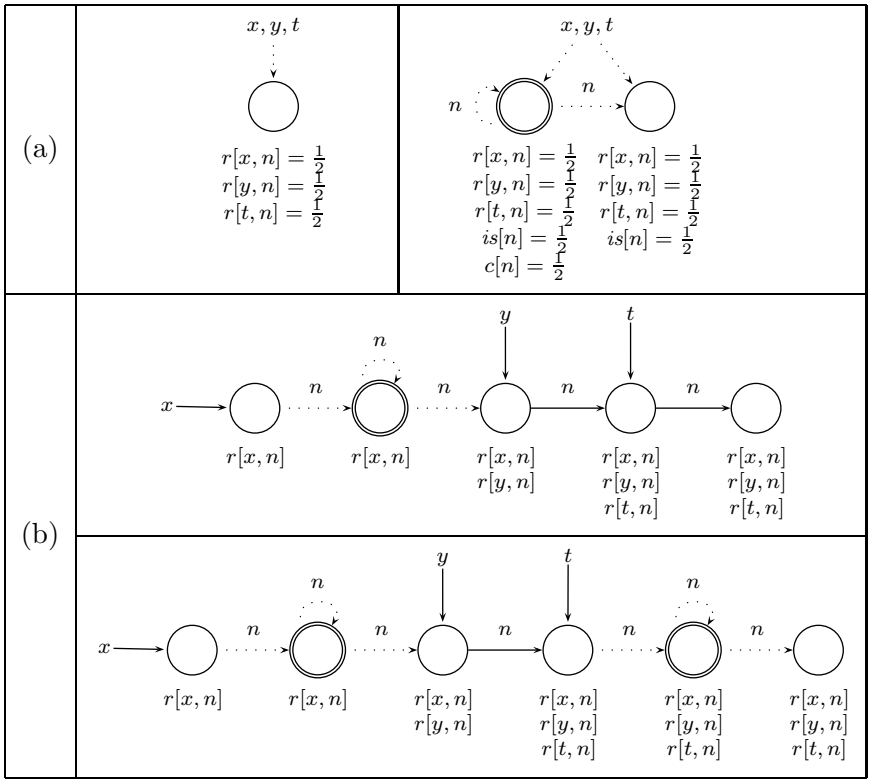

Figure 3.2: (a) a set of structures representing the semantic condition requiring the presence of a last element in a singly-linked list; (b) the set of structures obtained by applying meet on the set in (a) and the single structure in Fig. 2.1(b), and enforcing integrity constraints.

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.

Example 3.3. Fig. 3.2 shows an example of an abstraction refinement that can be obtained by a meet operator, but for whom the TVLA Focus is undefined (yielding an exception). The semantic condition used requires that a singly-linked list has a last element, leading to an infinite number of structures in [SRW02]. Setting $\widehat{XS}$ to be the set shown in Fig. 3.2(a), $\widetilde{\text{Focus}}_{\widehat{XS}}$ is applied to the structure shown in Fig. 2.1(b). The finite set of structures shown in Fig. 3.2(b) is that resulting from the meet-based focus operation. Note that, an additional step is taken in order to enforce *integrity constraints* on the resulted structures, thus sharpening imprecise structures and eliminating inconsistent ones, with respect to a set of constraints implied by the instrumentation predicates and accompanying integrity rules [SRW02].

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 [HRS95, DGS98] 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 [SRW02] yields an imprecise backward demand analysis. Primarily, this is due to the fact that the preciseness 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 preciseness 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.

Example 3.4. Fig. 3.3 demonstrates the use of a meet operator for enforcing feasibility constraints while performing backward demand shape analysis. Assume that, for some initial structure, the backward analysis yields the structure shown in Fig. 3.3(a) right before the statement x = y in line 16 of Fig. 1.1. The structures shown in Fig. 3.3(b)—as derived from the semantics of the statement y.n = x in line 15 of Fig. 1.1—represent the feasibility requirements that apply after that statement. Applying the meet operator to the structure sets in Fig. 3.3(a) and Fig. 3.3(b) yields the refined, feasible structure set shown in Fig. 3.3(c), right after the statement in line 15.

3.4 Interprocedural Analysis using Procedure Specific Abstractions

In [JLRS04], the meet operator is used to conduct functional interprocedural analysis (see [SP81]). Only the main ideas are sketched here.

Recall that the main problem in the functional approach to interprocedural analysis (and in structural dataflow analysis in general, e.g., [Tar81]) is operating on representations of sets of transitions between concrete states. The effect of a code block B is a binary relation $\tau \subseteq \mathcal{S}_2 \times \mathcal{S}_2$,

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

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

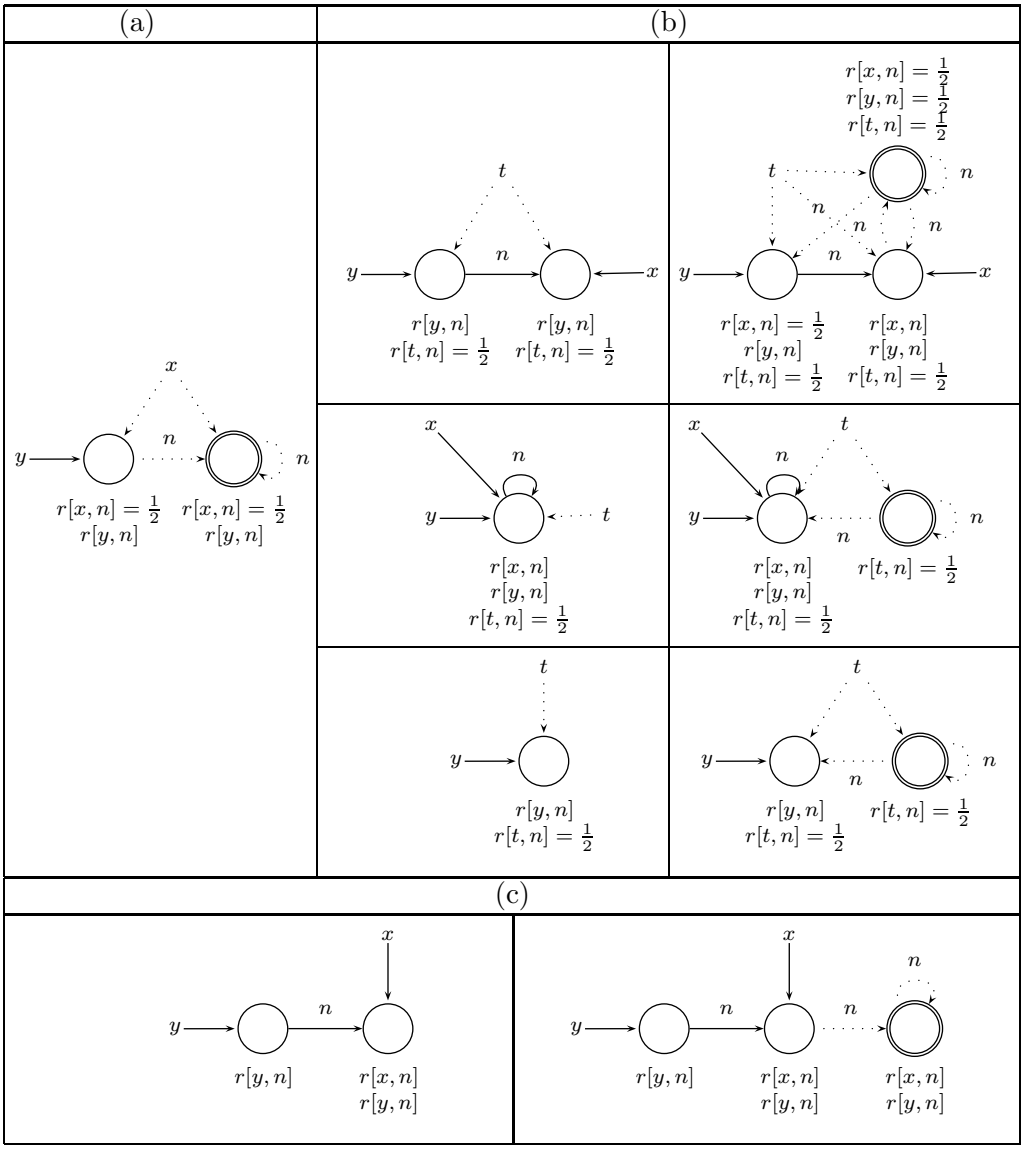

Figure 3.3: (a) a 3-valued structure representing a set of configurations that may arise after the statement y.n = x, as yielded by the non-deterministic update of the backward collecting semantics applied to the next statement, x = y; (b) the set of structures representing the feasibility condition associated with the statement y.n = x; (c) the resulted feasible structures as obtained by applying meet on (a) and (b).

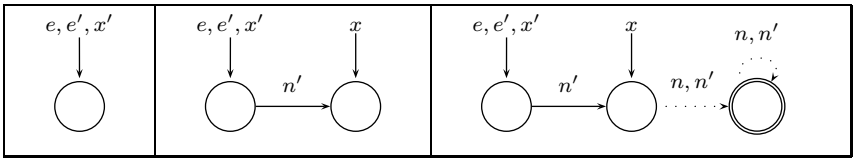

Figure 3.4: A set of dual-vocabulary, 3-valued structures representing the effect of x = prepend(e, x). Tagged predicates denote a posteriori properties.

states. The composition of τ_1 and τ_2 can be defined as

$$\left\{ (S_1, S_3) \;\middle|\; (S_1, S_2, S_3) \in \begin{array}{l} \left\{ (S_1, S_2, S') \,|\, (S_1, S_2) \in \tau_1,\ S' \in \mathcal{S}_2 \right\} \\ \quad \cap \left\{ (S'', S_2, S_3) \,|\, (S_2, S_3) \in \tau_2,\ S'' \in \mathcal{S}_2 \right\} \end{array} \right\} .$$

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. 3.4 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

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

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 $S[\mathcal{P}_{set} \leftarrow \frac{1}{2}]$, for $set \in \{in, out, tmp\}$, sets the predicates of $\mathcal{P}_{set}$ of the triple-vocabulary structure S to $\frac{1}{2}$.

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 $\frac{1}{2}$ 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 [Cou02, Theorem 13] it is shown how to employ abstract interpretation to an upper approximation to the (infinite) set of possible (infinite) traces. In [YRSW03], an abstract interpretation algorithm for computing such approximation was given. It represents traces using 3-values 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 details the reader is referred to [Cou02, 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 this point it is not used before being redefined.[1]

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

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

Here, it is assumed that the concrete 2-valued states 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, such that include all possible values associated with temporal properties. CS_{final} represents the set of states in which all temporal properties are set to their final values (that is, their values upon termination of the execution). We write $(l_1, l_2), S_{in} \overrightarrow{\rightsquigarrow} S_{out}$ 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 $(l_1, l_2), S_{out} \overleftarrow{\rightsquigarrow} S_{in}$ to denote the transformation induced by the

[1] This is somewhat similar to *persistent* [MP89] properties that continuously hold from a given point in the trace.

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 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 ($\sqcap$).

Example 3.5. Bidirectional staged analysis can be applied to obtain compile-time garbage collection information. 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 $(l_1, l_2), S_{out}{}^{\leftsquigarrow}S_{in}$ 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. 3.5(a) shows one of the structures that arise before the statement

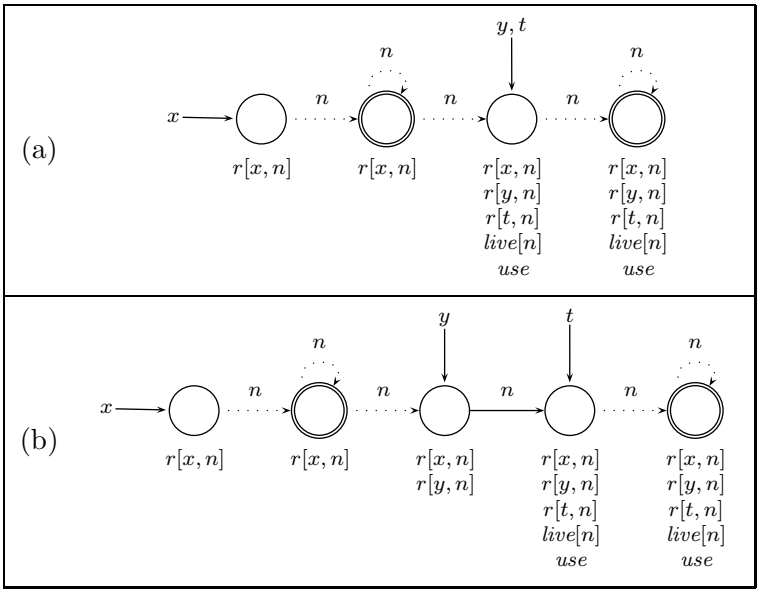

Figure 3.5: 3-valued structures representing sets of program configurations, including heap object and reference field liveness, that arise (a) before the execution of the statement t = y.n; and (b) after it is executed.

t = y.n at line 25 of Fig. 1.1, and Fig. 3.5(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 said 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 3.6. 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 [SKS01]. 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 $(l_1, l_2), S_{out} \overleftsquigarrow S_{in}$ 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. 3.5(a) shows one of the structures arising before the statement t = y.n at line 25 of Fig. 1.1, and Fig. 3.5(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.

Chapter 4

Computing Meet for Sets of Heap Abstractions

This section describes a proposed implementation of the meet operator for the powerset domain of 3-valued structures with Hoare ordering.

Since the applications concerned apply meet to a finite number of sets, it suffices to consider the problem of computing meet for two sets. The following lemma reduces the problem of computing the meet of two sets to the problem of computing the meet of a pair of single-element (singleton) sets.

Lemma 4.1. *Let X and Y be two elements of a partially ordered powerset with Hoare ordering. Then*

$$\bigsqcup_{\substack{x \in X \\ y \in Y}} \{x\} \sqcap \{y\} = X \sqcap Y \quad .$$

Proof. See Appendix A.1. □

Since the join operation is known to be efficiently computable for domains of canonic abstraction, in the rest of this section we only consider the following problem.

Problem 4.2. *Given two structures $S_1 = (U_1, I_1)$ and $S_2 = (U_2, I_2)$ over a fixed set of predicates $\mathcal{P}$, compute $\{S_1\} \sqcap \{S_2\}$.*

4.1 Computing Meet for a Pair of Abstract Heaps

We establish a connection between the structures that comprise the result of the meet operation and certain relations that hold between the input structures. We first define the meet of two Kleene values t_1 and t_2 by

$$t_1 \sqcap t_2 = \begin{cases} t_1 & \text{if } t_1 \sqsubseteq t_2, \\ t_2 & \text{if } t_2 \sqsubseteq t_1, \\ \bot & \text{otherwise.} \end{cases}$$

Definition 4.3 (Meet Correspondence). *Given two structures $S_1 = (U_1, I_1)$ and $S_2 = (U_2, I_2)$, a relation $M \subseteq U_1 \times U_2$ is a* meet correspondence *between S_1 and S_2 if it is (a)* full*, i.e.,*

$$u_1 \in U_1 \implies \exists v_2 \in U_2 : u_1 \, M \, v_2$$
$$v_2 \in U_2 \implies \exists u_1 \in U_1 : u_1 \, M \, v_2 \ ,$$

and (b) consistent*, i.e., for every predicate p of arity k, and a pair of k-tuples of nodes $u_1, \ldots, u_k \in {U_1}^k$ 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 4.4. *Given a meet correspondence M between two structures, $S_1 = (U_1, I_1)$ and $S_2 = (U_2, I_2)$, the operation $S_1 \sqcap_M S_2$ yields the M-*intermediate structure *$S = (U, I)$ of S_1 and S_2, where*

$$U = M \ ,$$

and the interpretation for any predicate p of arity k and any k-tuple of nodes $(u_1, v_1), \ldots, (u_k, v_k) \in U^k$ is given by

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

Clearly, from Definition 4.3 we get that an M-intermediate structure is well-defined for every meet correspondence M.

Proposition 4.5. *Given two structures $S_1 = (U_1, I_1)$ and $S_2 = (U_2, I_2)$, and a meet correspondence $M \subseteq U_1 \times U_2$, let $S = (U, I)$ be their M-intermediate structure obtained by $S_1 \sqcap_M S_2$. Then, $S \sqsubseteq S_1$ and $S \sqsubseteq S_2$.*

Proof. See Appendix A.2. □

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

Lemma 4.6. *Let $\mathcal{M}_{S_1,S_2} \subseteq \wp(U_1 \times U_2)$ denote the set of meet correspondences between two structures, $S_1 = (U_1, I_1)$ and $S_2 = (U_2, I_2)$. Then*

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

Proof. See Appendix A.3. □

Lemma 4.6 already gives us a naive algorithm to compute a meet value by enumerating all relations $M \in U_1 \times U_2$, and—for each of them which is a meet correspondence—compute $S_1 \sqcap_M S_2$. Unfortunately, this straightforward approach is not tractable, since possibly many of the $2^{|U_1| \times |U_2|}$ enumerated relations are not meet correspondences. Our proposed algorithm efficiently enumerates a superset of meet correspondences, but such that contains only very few redundant relations.

```
function MEET(S_1 = (U_1, I_1), S_2 = (U_2, I_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 quotas based on summary property. */
    Q_a, 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 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
```

Figure 4.1: An algorithm for computing the meet of structures S_1 and S_2 over a fixed a set of predicates $\mathcal{P}$.

4.2 Enumerating Meet Correspondences

An algorithm for computing the meet of two structures, $S_1 = (U_1, I_1)$ and $S_2 = (U_2, I_2)$, over a set of predicates $\mathcal{P}$, is shown in pseudo-code in Fig. 4.1. The algorithm views the problem of finding meet correspondences as finding constrained matchings in the complete bipartite graph $(U_1, U_2, U_1 \times U_2)$, such that satisfy the *fullness* and *consistency* constraints, as detailed in Definition 4.3. In order to find all constrained matchings, the algorithm uses an incremental approach to solve the constraints, as follows.

Verifying consistency of nullary predicates. A preliminary check is used to quickly determine whether the two structures disagree on a nullary predicate (in such a case the result of the meet is an empty set).

Solving consistency for unary properties. By unary property of an individual u, we consider the interpretation of any predicate of arity $k > 0$ over the k-tuple u^k. By checking every pair $(u, v) \in U_1 \times U_2$ for consistency over all unary properties, the algorithm avoids the formulation of redundant E edges: clearly, if for some $u \in U_1$ and $v \in U_2$, there exists a predicate p of arity $k > 0$ such that $p^{S_1}u^k \sqcap p^{S_2}v^k = \bot$, then (u, v) cannot be a part of any meet correspondence.

Solving fullness constraint. Observing that, in any meet correspondence—due to the special nature of the equality predicate *eq*—a non-summary individual of one structure can relate to at most one individual of the other structure, while every node must be matched at least once in order to satisfy the fullness constraint, the algorithm generates the following problem: given the edge set E formed in the previous stage, associate a matching range of $[1, 1]$ and $[1, \text{degree}(u, E)]$ to non-summary

and summary individuals, respectively. Then, find all matchings in the graph $(U_1 \cup U_2, E)$ that satisfy these quotas. Clearly, any such matching satisfies the fullness constraint. The solution to this problem is discussed in Section 4.3.

Solving remaining consistency constraints. This is achieved by checking consistency over all predicates of arity $k > 1$, for every matching computed by the previous stage.

Having formed and verified all meet correspondences incrementally, the algorithm then constructs and collects (join) all intermediate structures induced by those correspondences, as described in Definition 4.4.

4.3 Enumerating Full b-Matchings in a Bipartite Graph

We consider the following problem of graph matching.

Problem 4.7. *Given an undirected graph $G = (V, E)$ and minimal and maximal quotas on nodes $Q_a, Q_b : V \to \mathbb{Z}$, such that $Q_a(u) \leq Q_b(u)$ for all $u \in V$, find all $M \subseteq E$ such that $Q_a(u) \leq \left|\{(u, v) \in M\}\right| \leq Q_b(u)$ for all $u \in V$.*

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

Considering an input to Problem 4.7, we say that a node in the graph is *violated* if its lower quota exceeds its degree, thus such quota cannot be satisfied by any matching. We say that a node is *saturated* if its upper quota 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 4.8. *Given an input to Problem 4.7, 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 quotas of both endpoints by 1, solve the new problem, and add e to any matching in the returned solution.*

Then, applying this strategy yields a solution to the problem.

Proof. See Appendix A.4. □

While an algorithm to enumerate generalized matchings can be immediately derived from the strategy described in Lemma 4.8, 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 4.9. *Given an input to Problem 4.7, we define the* residual combinatorial degree *of a node u to be*

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

Put in another way, the RCD of a node, with respect to an edge set and lower and upper matching quotas, is the number of subsets of its incident edges set that satisfy its quotas. In the case of ordinary matching, where both quotas 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 quota 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 quotas.

```
function EnumAbMatch((V, E), Q_a, Q_b)
    /* If graph 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 / saturated, throw 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 throw it from 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 quotas, 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
```

Figure 4.2: An algorithm for enumerating all matchings in a graph with lower and upper node matching quotas.

Fig. 4.2 shows the pseudo-code of our proposed heuristic algorithm, which applies the minimal RCD heuristic on node selection of the strategy described in Lemma 4.8. The correctness of this algorithm is immediately inherited from that of its underlying strategy.

4.4 Correctness

The following theorem establishes the correctness of the meet algorithm described in Section 4.2.

Theorem 4.10. *Given an input to Problem 4.2, the result of* $\textsc{Meet}(S_1, S_2)$ *is a correct solution.*

Proof. As the algorithm joins M-intermediate structures for all $M \in \mathcal{M}_{S_1,S_2}$, following Lemma 4.6 it yields $\{S_1\} \sqcap \{S_2\}$. □

4.5 Performance

In the general case, the computation of meet for a pair of arbitrary 3-valued structures might yield a result that is exponential in the size of the input, due to combinatorial explosion induced by $\frac{1}{2}$ predicate values. Hence, the efficiency of an algorithm to compute meet for a 3-valued structure powerset domain is measured with respect to the size of the output. But, even so it appears that polynomial bounds by the size of the output can not be proved: a polynomially bound solution to the 3-valued structure meet problem could easily be reduced to solve the 3-colorability decision problem in polynomial time, thus is NP-hard. Note that, this is in line with (and an immediate consequence) of [Yor03].

Nonetheless, as the surveyed applications in general—and the experimental part concerning bidirectional staged analysis in particular—mostly

manipulate canonic (or near-canonic) bounded structures that are proper α-images of their represented concrete structures, the enumeration of meet correspondences is simplified: unary abstraction predicate values determine strict correspondence between non-summary individuals, thus their matching induces no degree of freedom through the search procedure. Since, thanks to the heuristic used, those are matched earlier through the search, it follows that the expansion of the search space due to matching alternatives for summary individuals is postponed to the tail of the recursive search process, thus keeping the search tree mostly linear and minimizing redundant paths.

Indeed, the effectiveness of the heuristic approach has been empirically tested in several settings. More specifically, its performance was measured with respect to two sources of redundancy that might occur during the computation:

1. The percentage of non-meet-correspondences—such that do not satisfy consistency *for all* tuples of individuals over all predicates—out of all unary-based correspondences, as computed by EnumAbMatch. This indicates the level of redundancy involved with the computation of correspondence relations that is based on unary property inconsistency elimination only.

2. The percentage of non-beneficial recursive steps—whose returned value is an empty set—out of total recursion steps. This indicates the effectiveness of the heuristic approach that is applied to the enumeration strategy.

Fortunately, throughout gained experimental experience, both ratios are very close to 0 (see Section 5). This suggests that the described algorithm

performs virtually polynomial by the size of the output, with respect to some empirically obtained redundancy factor. The exact computational overhead is determined by the actual implementation choices taken, specifically those used for solving full b-matchings enumeration, as follows:

1. Priority queues, used to maintain order and obtain nodes with minimal RCD values, as well as supporting an operation of RCD value key decrement upon decreasing of node quotas and edge removals.

2. The actual computation induced by RCD value updates.

Alternatives for both selections, as well as details of the actual implementation chosen and possible performance improvements, are further discussed in Appendix B.

Finally, the proposed algorithm also outperforms several other implementations of the meet operator, as described in Section 6.

Chapter 5

Applications and Experimental Results

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 approach, the former aimed at allowing compile-time garbage collection and the latter promoting earlier reclaiming 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.

The instantiation of those analyses leans on an implementation of the proposed meet algorithm, as well as additional auxiliary mechanism, in TVLA [LAS00]. These are described in Appendix B.

5.1 Free Analysis

The analysis described in this section aims at providing for 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 intolerable due to limited space and processing power. Such platforms normally support an explicit **free** directive, and require static deallocation schemes for dynamic memory objects.

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 3.5. The first (forward) stage tracks shape related information (see Table 2.1) but keeps the $use(v)$ predicate value to be $\frac{1}{2}$ 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 $\frac{1}{2}$ 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. However, in order to avoid redundant **free** statements (e.g., generating a sequence of **free** statements all along some objects "death" period), this information is only generated for particular program locations, where usage information is known to be affected, namely such involving the evaluation of a dereference expressions. Note that such a simple heuristic in conservative, as it might miss locations that are verified for a safe **free** statement where particular branch involving patterns are involved. Nonetheless, a more rigorous approach—namely, such that examines the relations between pairs of

consecutive program locations around a conditional statement—can easily overcome this shortcoming.

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 [SKS01] 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 3.6. Similar to the above (see Section 5.1), $live[f](v)$ predicate values are retained $\frac{1}{2}$ (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. Similarly,

a heuristic is applied to yield only these locations where the liveness of a reference field may be affected. Similar drawbacks and possible compensation hold for this case as well.

5.3 Prototype Implementation

The implemented prototype is derived from the one described in [SYKS03], and is based on meet capable version of the TVLA framework [LAS00] (see Appendix B). The prototype consists of the following components:

A front-end, which translates a Java program (.class file) into a TVLA program. It is based on the J2TVLA translation engine, written by Roman Manevich, and uses the Soot framework [VRHS+99].

A bidirectional staged analyzer, which applies either of the above described analyses to the translated TVLA program. A dedicated specification is used to instantiate the appropriate static analysis algorithm, as per the required analysis.

A back-end, which traverses the analysis results and extracts free or nullify information, respectively. It is implemented using TVLA libraries.

5.4 Experimental Results

Table 5.1 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 [SKS01], 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.

Program	Description
Loop	Construct and traverse a linked list (Fig. 1.1)
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 5.1: description of the benchmark programs.

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. 1.1, 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 5.2 shows the costs of the analyses as they were applied to the benchmark programs. It only considers the core analysis done by TVLA, as front-end and back-end computational overhead is insignificant compared to

Program	Forward		Backward		Meet redundancy %			
	Time	Space	Time	Space	Matching		Recursion	
Free analysis								
Loop	4.14	1.20	8.80	4.42	0.0	(0.0)	2.0	(0.7)
CReverse	9.42	2.64	22.57	12.02	0.0	(0.0)	1.4	(0.5)
Delete	28.50	5.71	119.89	18.90	0.8	(0.02)	1.1	(0.05)
DLoop	6.82	1.72	13.20	8.11	0.0	(0.0)	2.3	(0.6)
DPairs	14.47	2.93	20.83	11.20	0.0	(0.0)	1.1	(0.3)
small−javac	656.75	26.18	543.17	51.33	17.0	(0.6)	5.9	(0.5)
Nullify analysis								
Loop	4.00	1.21	9.21	4.40	0.0	(0.0)	1.7	(0.7)
CReverse	9.71	2.73	22.61	11.59	0.0	(0.0)	1.4	(0.5)
Delete	28.10	5.82	119.32	19.13	0.7	(0.02)	1.1	(0.06)
DLoop	6.71	1.74	12.12	8.16	0.0	(0.0)	1.3	(0.6)
DPairs	14.58	3.20	25.79	11.66	0.0	(0.0)	0.9	(0.2)
small−javac	607.44	27.19	573.58	52.87	17.6	(0.7)	6.2	(0.5)

Table 5.2: analyses costs for the benchmark programs. Time is measured in seconds, and space is measured in MB. Meet redundancy shows the percentage of redundant computations, denoting both overall and average case (in parentheses), w.r.t. redundant enumerated matchings as well as redundant recursion steps through matchings enumeration.

that of TVLA. The experiments were done on an AMD Athlon XP 2800 machine with 512 MB of memory, running Windows XP. Several observations can be made:

- Obviously, time and space consumption is strictly correlated between the two analyses, as they both use very similar abstractions (only the object use and field liveness predicates differ), and consequently their induced transition formulas resemble.

- In most cases, the backward stage takes significantly more time compared to the forward stage (up to a factor of 4 for the Delete program). This can be attributed to the naive implementation of the meet algorithm, which uses explicit recursion for graph matchings enumeration, thus causing excess overhead as whole graphs are being duplicated through recursive calls. Indeed, analysis statistics indicate that the meet operator has the lion's share concerning this increase in time (taking up to 70% of the backward analysis time for the Delete program). This also explains the excess memory consumption experienced in the backward stage. (See Appendix B for a discussion of implementation alternatives.)

- Redundancy measurement results of the meet algorithm (following Section 4.5) indicate that—in most benchmark programs—the overall percentage of non-meet-correspondence enumerated matchings is kept below 1% (and is literally zero for most cases), and that the percentage of non-beneficial recursive steps is kept below 2.5% (with average case below 1%). For the small−javac case, the analyses suffer from significantly greater overall redundancy (up to 17.6% non-meet-correspondence matchings, and up to 6.2% redundant recursion

steps). However, the low average case measures (below 0.7% non-meet-correspondences, and 0.5% redundant recursion steps), imply that worst-case redundant cases are singular and happen quite rarely. These results strengthen our intuition that the proposed heuristic indeed promotes fast convergence of meet correspondences enumeration, and is highly suitable for this kind of analyses.

- While the variance among analyses times of the benchmark programs generally corresponds to that experienced in earlier work [SYKS03], where single queries were manually issued to the same benchmark programs, it appears that time overhead has increased by an order of magnitude. As this unfortunate performance degradation is unexpected—analysis complexity has been reduced by most aspects compared to the above work—we believe that it can be attributed to the changes that the TVLA system has gone through since then, such as the use of differencing to automatically update instrumentation predicates.

Hence, it is evident that our meet algorithm is practically highly efficient, in the asymptotic sense, as it induces very low redundancy rates compared to the actual size of the output. Also, it is shown that a meet-based, bidirectional staged analysis is capable of capturing precise heap object use and field liveness information, hence is suitable for automatic discovery of garbage collection information during compile-time. Nonetheless, the current implementation involves considerable computational overhead, partly due to circumstantial implementation choices in the meet algorithm (that can mostly be overridden by alternative means), as well as performance and scaling problems that are framework inherent.

Chapter 6

Related Work

6.1 Computing Meet of Heap Abstractions

In [JLRS04], a meet operator is used to instantiate 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,[1] 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 triv-

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

ial 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 [JLRS04] 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 [KR04], 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 [JLRS04]: first, a normalization step is applied to the formulas, similar to the canonicalization step in [JLRS04]; then, conjunction is used to compute the result.

In [YRS04], 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

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

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 Enumerating Matchings in Graphs

The problem of finding matchings in graphs has been extensively studied over the years. In particular, the generalized variant of finding a b-matching, given some quota function $b : V \to \mathbb{N}$ on nodes, can be modeled by a constrained class of integer linear programming problems, such that can be solved in polynomial time [EJ70]. Clearly, the problem of finding a matching such that satisfies lower node quotas as well, can be reduced to b-matching in a straightforward manner, by doubling the number of inequalities.

This implies that the enumeration of all applicable b-matchings can be obtained as a simple application of the above: given an input to the matching problem, start by obtaining two distinct valid matchings. Then, for some $e \in E$ that distinguishes between them, recursively find all valid matchings including and excluding e. As each recursive step is guaranteed to run in polynomial time and yield at least one valid matching, the total time required for this algorithm is guaranteed to be polynomial by the size of the output.

Obviously, since polynomial time bounds were not proved for the matching algorithm presented in this thesis, such an approach is favorable when

asymptotic time bounds are concerned. Nonetheless, with practical considerations in mind, our described enumeration method seems to be beneficial as it does not involve exhaustive generation of independent valid matchings, but rather the incremental generation of the whole set. Our experience shows that redundancy factors induced by this algorithm are very low, implying very good behavior in practice.

6.3 Compile-Time Memory Management Analysis

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 [Bar77, ISY88, FW91, Ham95, Jon99]. This thesis 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., [Bla98]) 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 [BTV96, TT94, AFL95, HET02], 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 [Muc97] 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 [App92, ADM98, HDH02], liveness information for root references is used to promote unused space reclamation. In [SKS02], 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 thesis, using a staged static analysis algorithm.

In [SYKS03], 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 [SYKS03] 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 thesis 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, it is believed that this approach may be significantly more efficient compared to a multiple query instantiation of the analysis in [SYKS03].

Chapter 7

Conclusion

In this thesis 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.

Our results suggest several directions for future work. First, the implementation of the meet algorithm can be significantly improved to minimize processing overhead due to explicit recursion and excess duplication of data structures. This, along with further improvements to the TVLA framework, may lead to better scalability of our proposed compile-time GC mechanisms for larger programs, as well as scaling other techniques, such as interprocedural shape analysis. Unfortunately, we failed to apply our compile-time

analyses to actual JavaCard example programs (as it is done in [SYKS03]) and upcoming efforts are hoped to achieve this goal. Finally, further refinements to our method of computing meet, as well as possible concrete asymptotic time bounds for certain cases of bounded 3-valued structures, can be pursued.

Bibliography

[ADM98] 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.

[AFL95] 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.

[App92] Andrew W. Appel. *Compiling with Continuations*, chapter 16, pages 205–214. CUP, 1992.

[Bar77] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, 1977.

[Bla98] Bruno Blanchet. Escape analysis for object oriented languages. Application to Java™. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, pages 20–34. ACM Press, 1998.

[BO96] Gerth Stølting Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–857, 1996.

[BTV96] 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.

[CC77] 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.

[Cou02] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comp. Sci.*, 277:47–103, April 2002.

[DGS98] 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.

[EJ70] Jack Edmonds and Ellis L. Johnson. Matching: a well-solved class of integer linear programs. In *Combinatorial Structures and their Applications, Calgary International Conference*, pages 89–92. Gordon and Breach, 1970.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[FW91] Ian Foster and William Winsborough. Copy avoidance through compile-time analysis and local reuse. In *Proceedings of Inter-*

national Logic Programming Sympsium, pages 455–469. MIT Press, 1991.

[Ham95] 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.

[HDH02] 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.

[HET02] 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.

[HRS95] 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.

[ISY88] 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.

[JLRS04] 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.

[Jon99] Richard Jones. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1999.

[Kil73] 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.

[KR04] 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.

[KRS94] 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.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[LAS00] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *Proc. Static Analysis Symp.*, pages 280–301, 2000.

[MP89] 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.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[SKS00] 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.

[SKS01] 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.

[SKS02] 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.

[SP81] 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.

[SRW02] 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.

[SYKS03] 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.

[Tar81] Robert E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, 1981.

[TT94] 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.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot—a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[Yor03] Greta Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2003. http://www.cs.tau.ac.il/∼gretay/.

[YRS04] 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.

[YRSW03] Eran Yahav, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying temporal heap properties specified via evolu-

tion logic. In *European Symposium on Programming Languages (ESOP)*, 2003.

Appendix A

Proofs

Recall the definition of the Hoare ordering for two sets X and Y to be

$$X \sqsubseteq Y \iff \forall x \in X \; \exists y \in Y : x \sqsubseteq y \ . \tag{A.1}$$

Also, for some poset, recall that $Z = X \bigsqcap Y$ if Z is a lower bound of X and Y

$$Z \sqsubseteq X \wedge Z \sqsubseteq Y \ , \tag{A.2}$$

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

$$Z' \sqsubseteq X \wedge Z' \sqsubseteq Y \implies Z' \sqsubseteq Z \ . \tag{A.3}$$

A.1 Proof of Lemma 4.1

Proof. We show that Z is the meet of X and Y, by proving compliance to both Eq. (A.2) and Eq. (A.3).

From the definition of meet, for some $x \in X$, $y \in Y$ it holds that $\{x\} \sqcap \{y\} \sqsubseteq \{x\}$ and $\{x\} \sqcap \{y\} \sqsubseteq \{y\}$. Since $\{x\} \sqsubseteq X$ and $\{y\} \sqsubseteq Y$, from transitivity of the partial order it follows that $\{x\} \sqcap \{y\} \sqsubseteq X$ and $\{x\} \sqcap \{y\} \sqsubseteq Y$. Considering the set

$$\big\{ \{x\} \sqcap \{y\} \,\big|\, x \in X, y \in Y \big\} \ ,$$

it holds that both X and Y are among its upper bounds. As Z is defined to be its least upper bound, we get that $Z \sqsubseteq X$ and $Z \sqsubseteq Y$, thus satisfying Eq. (A.2).

Let Z' be some Hoare poset element such that $Z' \sqsubseteq X$ and $Z' \sqsubseteq Y$. From Eq. (A.1) we get that

$$\forall z' \in Z' \, \exists x \in X : z' \sqsubseteq x$$

and

$$\forall z' \in Z' \, \exists y \in Y : z' \sqsubseteq y \ .$$

The combination of the above yields

$$\forall z' \in Z' \, \exists x \in X, y \in Y : z' \sqsubseteq x \wedge z' \sqsubseteq y \ ,$$

and applying Eq. (A.1) we get that

$$\forall z' \in Z' \, \exists x \in X, y \in Y : \{z'\} \sqsubseteq \{ x\} \wedge \{ z'\} \sqsubseteq \{y\} \ .$$

From the definition of meet it follows that

$$\forall z' \in Z' \, \exists x \in X, y \in Y : \{z'\} \sqsubseteq \{ x\} \sqcap \{y\} \ .$$

Clearly, for any $x' \in X$, $y' \in Y$ it holds that

$$\{x'\} \sqcap \{y'\} \sqsubseteq \bigsqcup_{\substack{x \in X \\ y \in Y}} \{x\} \sqcap \{ y\} = Z \ ,$$

hence, from transitivity of the partial order relation it follows that

$$\forall z' \in Z' : \{z'\} \sqsubseteq Z \ .$$

From Eq. (A.1) we get that

$$\forall z' \in Z' \, \exists z \in Z : z' \sqsubseteq z \ ,$$

and applying it again yields $Z' \sqsubseteq Z$, thus satisfying Eq. (A.3). $\square$

A.2 Proof of Proposition 4.5

Proof. We form a pair of functions $f_1 : U_1 \to U$ and $f_2 : U_2 \to U$, that embed S into S_1 and S_2, respectively. Let

$$f_1 = \left\{ \big((u,v),u\big) \;\middle|\; (u,v) \in U \right\}$$

and

$$f_2 = \left\{ \big((u,v),v\big) \;\middle|\; (u,v) \in U \right\} \ .$$

Following Definition 4.3, M is a *full* relation. Therefore, from the construction of S as in Definition 4.4, if follows that both f_1 and f_2 are surjective. Additionally, for any predicate p of arity k and any k-tuple of nodes $(u_1,v_1),\ldots,(u_k,v_k) \in U^k$, it holds that

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

and

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

From the definition of f_1 and f_2 we get

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

and

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

Hence, $S \sqsubseteq S_1$ and $S \sqsubseteq S_2$. □

A.3 Proof of Lemma 4.6

Proof. We show that XS is the meet of $\{S_1\}$ and $\{S_2\}$ by proving it satisfies Eq. (A.2) and Eq. (A.3).

From Proposition 4.5, for any $M \in \mathcal{M}_{S_1,S_2}$ and an induced M-intermediate structure S, it holds that $S \sqsubseteq S_1$ and $S \sqsubseteq S_2$. Then, considering the set

$$\left\{\{S_1 \sqcap_M S_2\} \mid M \in \mathcal{M}_{S_1,S_2}\right\}$$

it holds that both $\{S_1\}$ and $\{S_2\}$ are among its upper bounds. As XS is defined to be the least upper bound of this set, we get that $XS \sqsubseteq \{S_1\}$ and $XS \sqsubseteq \{S_2\}$, thus satisfying Eq. (A.2).

Let XS' be some 3-valued structure set such that $XS' \sqsubseteq \{S_1\}$ and $XS' \sqsubseteq \{S_2\}$. From Eq. (A.1) we get that

$$\forall S' \in XS' : S' \sqsubseteq S_1$$

and

$$\forall S' \in XS' : S' \sqsubseteq S_2 \ .$$

The combination of the above yields

$$\forall S' \in XS' : S' \sqsubseteq S_1 \wedge S' \sqsubseteq S_2 \ .$$

Let $S' = (U', I')$ be some structure in XS', then there exist surjective functions $f_1 : U' \to U_1$ and $f_2 : U' \to U_2$, that embed S' into S_1 and S_2, respectively. Let $M = f_2 \circ {f_1}^{-1} \subseteq U_1 \times U_2$. Since both f_1 and f_2 are surjective, M is *full*. Also, for any predicate p of arity k, and any k-tuple $w_1, \ldots, w_k \in {U'}^k$,

$$p^{S'}(w_1, \ldots, w_k) \sqsubseteq p^{S_1}(f_1(w_1), \ldots, f_1(w_k))$$

and

$$p^{S'}(w_1,\ldots,w_k) \sqsubseteq p^{S_2}(f_2(w_1),\ldots,f_2(w_k)) \ ,$$

therefore

$$p^{S'}(w_1,\ldots,w_k) \sqsubseteq p^{S_1}(f_1(w_1),\ldots,f_1(w_k)) \sqcap p^{S_2}(f_2(w_1),\ldots,f_2(w_k)) \ . \tag{A.4}$$

Since for any such predicate and tuple $p^{S'}(w_1,\ldots,w_k) \neq \bot$, it holds that

$$p^{S_1}(f_1(w_1),\ldots,f_1(w_k)) \sqcap p^{S_2}(f_2(w_1),\ldots,f_2(w_k)) \neq \bot \ .$$

Let $u_1,\ldots,u_k \in {U_1}^k$ and $v_1,\ldots,v_k \in {U_2}^k$ be two k-tuples, such that $u_i \, M \, v_i$ for $i = 1 \ldots k$. From the construction of M, there exist a k-tuple $w_1,\ldots,w_k \in {U'}^k$, such that $u_i = f_1(w_i)$ and $v_i = f_2(w_i)$ for $i = 1 \ldots k$, hence

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

namely, M is *consistent* and therefore is a meet correspondence.

Let $S = (U, I)$ be the M-intermediate structure $S_1 \sqcap_M S_2$. From the construction of M and Definition 4.4, we get that

$$U = \left\{ (f_1(w), f_2(w)) \,\middle|\, w \in U' \right\} \ .$$

Define $f : U' \to U$ to be

$$f = \left\{ \big(w, (f_1(w), f_2(w))\big) \,\middle|\, w \in U' \right\} \tag{A.5}$$

Clearly, f is a surjective function. Following Definition 4.4, for any predicate p of arity k, and any k-tuple $((f_1(w_1), f_2(w_1)),\ldots,(f_1(w_k), f_2(w_k))) \in U^k$ (alternatively, for any k-tuple $w_1,\ldots,w_k \in U'$),

$$p^S((f_1(w_1), f_2(w_1)),\ldots,(f_1(w_k), f_2(w_k))) = \\ p^{S_1}(f_1(w_1),\ldots,f_1(w_k)) \sqcap p^{S_2}(f_2(w_1),\ldots,f_2(w_k)) \ .$$

Then, following Eq. (A.4), for any such predicate and tuple, it holds that

$$p^{S'}(w_1, \ldots, w_k) \sqsubseteq p^{S}((f_1(w_1), f_2(w_1)), \ldots, (f_1(w_k), f_2(w_k))) \ .$$

Then, following Eq. (A.5),

$$p^{S'}(w_1, \ldots, w_k) \sqsubseteq p^{S}(f(w_1), \ldots, f(w_k)) \ ,$$

and so f embeds S' into S, thus $S' \sqsubseteq S$. Since XS is the least upper bound of all $\{S_1 \sqcap_M S_2\}$, $M \in \mathcal{M}_{S_1,S_2}$, there exists some $S^* \in XS$ such that $S \sqsubseteq S^*$. Transitivity of the partial order relation yields $S' \sqsubseteq S^*$, $S^* \in XS$, for any $S' \in XS'$. It follows that $XS' \sqsubseteq XS$, thus satisfying Eq. (A.3). □

A.4 Proof of Lemma 4.8

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. □

Appendix B

Implementation

This section describes the implementation of the meet algorithm and its integration into the TVLA framework of [LAS00]. Some technical improvements over the schematic description in Section 4 are briefly discussed. Finally, guidelines for applying this framework to obtain bidirectional staged analysis of heap-manipulating programs are sketched.

B.1 Meet algorithm implementation

The binary meet operator for 3-valued structure powerset elements has been implemented in the TVLA framework. The implementation is immediately derived from the pseudo-code shown in Fig. 4.1 and Fig. 4.2, together with the observation of Lemma 4.1: a wrapper method collects (join), for each pair of structures of the operand sets, the respective set of structures composing their meet value; this set is computed by an inner method, as described in Fig. 4.1, which in turn uses a helper call to enumerate the set of constrained matchings over the bipartite graph representing the unary correspondence edges, as described in Fig. 4.2.

In order to measure the effectiveness of our heuristic approach, collected

statistics are used to record redundant computation. These follow the guidelines described in Section 4.5.

Several low-level details that are hidden in the pseudo-code description of the matching algorithm should be addressed by a proper implementation. One of them is a proper way to compute RCD values. For that matter, the following easily computable alternative definition of RCD was used:

$$\mathrm{RCD}(u, E, Q_a, Q_b) = \sum_{d=Q_a(u)}^{Q_b(u)} \binom{\mathrm{degree}(u, E)}{d} \ .$$

Note that, in this context, for some $n \geq 0, i > 0$, we naturally define $\binom{n}{0}$ to be 1, $\binom{n}{n+i}$ and $\binom{n}{-i}$ to be 0.

The repeated selection of a node of least RCD value, while continuously decreasing RCD values due to edge selection and node quota decrements, suggests the use of a priority queue with an efficient decrease-key function. In practice, a Fibonacci heap [FT87] was used.

Some further improvements were considered but not yet implemented. The most important of them is the development of an iterative variant to the recursive declarative description. Obviously, the use of literal recursion induces excess duplication of data structures between recursion steps, as well as extra overhead caused by the recursive calls themselves. The porting into an iterative version requires some "generational notion" to be supported by the priority queue. One way to achieve this is use a fully persistent priority queue with minimal (constant) space penalty and similar time bounds, such as the purely functional priority queues of [BO96].

Another technical improvement addresses the actual decrement of RCD values, by exploiting Pascal triangle dependencies to reduce the amount of binomial coefficient calculations down to a constant number, per recursion

step.

B.2 Reference set functionality

Aiming at easy means to apply staged analysis of heap-manipulating programs, and in particular a bidirectional analysis as described in Section 3.6, simple functionality has been provided in TVLA, allowing the reuse of results of a previous analysis stage for the purpose of abstraction intersection during a subsequent stage.

First, a simple mechanism was provided to load the results of a previously concluded analysis, for each analysis node (e.g., program location), as a special *reference* member associated with that node. Second, the analysis sequence for each analysis node has been extended such that, in addition to the normal transformer that is applied to each input element, a binary meet is performed against the target node's reference set, prior to merging (join) the resulted structures into that node's working set. In particular, for a bidirectional analysis as described in Section 3.6, this refinement of abstraction through meet is the abstract equivalent of the set intersection applied by the backward collecting semantics.

The above mechanism can be easily adopted to instantiate a bidirectional staged heap analysis, by following these simple guidelines (note that, heap reference liveness analysis is used as example):

- A forward analysis stage employs transformers that update shape-related predicates, while keeping all liveness-related predicates as $\frac{1}{2}$. The results of this stage are saved for future use.

- A backward analysis stage first loads the results of the above forward stage. Starting with liveness predicate values set to 0 (assuming ev-

erything is dead by the end of the execution) and unknown $\frac{1}{2}$ shape values, it performs a preliminary intersection with the reference set of the exit node, to get the actual initial abstraction for the backward stage.

- The backward analysis' transformers update liveness-related predicates, while conservatively setting shape-related predicates to $\frac{1}{2}$ when a destructive update is encountered. Nonetheless, since a meet operator is applied against the target node's reference prior to merging the results into that node, the overall resulting abstraction is precise both liveness- and shape-wise.

An instantiation of such a staged analysis is evaluated in Section 5.