

# Improving Scalability of 3-Valued Logic Shape Analysis by Loosening Embedding

Gilad Arnold

January 18, 2006

## Abstract

Making precise shape analysis frameworks scale computationally for real-life programs is highly desirable, yet far from trivial. In this work we describe a technique that has been applied to a 3-valued logic based shape analysis framework, in attempt to pursue this goal. As opposed to earlier work, in which we focused on improving performance of the analysis through optimizing abstraction domain operators and introducing operator-based abstract transformer, this paper explores a different direction, in particular aiming at reducing the abstract state-space inflation, the root cause for unscalability of the analysis. By redefining the structure ordering (embedding) relation underlying heap state abstraction, we allow trimming of abstract elements which represent “special cases”, in the presence of an element which represents a “general case”. We achieve this by allowing heap summary nodes to represent *zero or more* nodes, and also extending the notion of binary predicate embedding and quantifier interpretation, accordingly. As these extensions preserve the abstraction invariants, proving the soundness of the resulting analysis is straightforward. Although implying additional algorithmic overhead, as well as a possible loss of precision, initial experimental results suggest that the extended framework achieves more compact and faithfully descriptive abstractions of program heap states, and furthermore manages to shrink the abstract powerset elements by a factor that is covariant with the size and complexity of the program, specifically up to 60%, cutting the analysis time by up to 55%, for a small set of micro-benchmarks.

## 1 Introduction

The ability to reason about the set of heap configuration that a computer program may exhibit, without actually running the program, has many applications in static program analysis: these include whole-program verification tasks like verifying the absence of null dereferences,

proving the correctness of heap intensive algorithms like garbage collectors, and reasoning about properties of heap references through the program (e.g., inferring dead objects, or object reference fields, with applications to static garbage collection [2]). Shape analysis also generalizes alias analysis, and therefore may replace traditional techniques such as pointer analysis, with intention of improving on the precision of the resulting optimizations. Nonetheless, shape analysis appears to be among the hardest problems in static program analysis: as it is undecidable to prove even simple properties of very small programs manipulating dynamic data structures, with pointers and destructive updates, even the compulsory use of conservative abstraction methods following [3] implies non-trivial frameworks, which in turn induce considerable complexity issues. Sources for such issues include the size of an abstract domain of this kind, as well as the complexity of the algorithms used for implementing transformers, abstraction and concretization, and various domain operators. In the following, we elaborate on some aspects of this problem, and suggest techniques that may reduce their impact.

In this work, we consider a framework for conducting static shape analysis, which models heap topology and related properties using logical structures, and applies first-order logic formulas to model the semantics of program transformations [5]. For the purpose of abstraction, (potentially unbounded) sets of (potentially unbounded) logical structures occurring during the analysis of some program, that have similar properties as per the analysis in question, are collapsed into finite, abstract “representative” structures, while keeping these properties intact. Although analyses instantiated by this framework are proven to yield precise and meaningful results compared to the actual (concrete) set of configurations a program exhibits [4], it is not so widely studied, let alone deployed in actual production-level compilers or analysis tools, mainly because it is considered a computational dead-end for most practical purposes: while the TVLA [4] reference implementation was used to demonstrate the precision of such

```

x = null;
while (...) {
    y = new SLL();
    ...
    y.n = x;
    x = y;
}

y = x;
while (y != null) {
    ...
    t = y.n;
    y = t;
}

```

Figure 1: A simple Java program that constructs a singly-linked list, then traverses its element from head to tail.

analysis, as well as the adaptivity of the 3-valued logic canonical abstraction domain for a wide variety of shape-related problems, it could not effectively scale for programs beyond a few hundreds of code lines. The reasons for this can be roughly split into two separate issues: first, a significant portion of the analysis time is due to particular algorithms that are being used through the abstraction and for implementing the various transformers, including implementation-specific choices made in the construction of the framework. The second issue—and probably the more fundamental one—is the huge abstract domain underlying the analysis, whose induced worst-case complexity is double-exponential by the number of abstraction predicates (essentially, the number of reference variables in the program). Yet, even when the analysis reaches a “graceful” and precise fixed point, it may have tens of abstract elements associated with each program location, even for simple and fairly small benchmarks.

Figure 1 shows a simple program that manipulates a singly-linked list. An automatically generated CFG representation of this program has 33 nodes, nonetheless, analyzing it using a standard heap shape abstraction [1] yields a total of 109 abstract heap structures. For a slightly more complicated program, that has 3 traversal loops and whose intermediate representation has 49 nodes, our analysis yields a total of 573 abstract heap descriptors. This demonstrates the steep state-space inflation that we experience when applying shape analysis to programs of increasing complexity.

In recent work [1] we have shown that a fresh and restricted implementation of the 3-valued analyzer can improve on the performance limitations of the reference im-

plementation, by taking a different approach for transformer implementation, as well as using specialized data structures along with optimized algorithms for implementing domain operators. By that, we have tackled issues related to the first above mentioned problem, but it was pretty evident that any improvement of this kind cannot address the core problem, namely reducing the state space inflation.

This paper describes an attempt to attack one aspect of the state explosion problem with 3-valued logic shape analysis. The major contributions of this work seem to be the following.

- We observe that certain code patterns—specifically, loops for iterating over recursive data structures—might yield abstract elements that are expressively redundant, in the sense that one of them (the “general case”) can be used to describe the others (“special cases”), as the latter do not offer significant precision over the former. This occurrence is due to the definition of the embedding relation among 3-valued structures, which is used to determine redundancy within powerset abstract elements, in turn defining the abstract sub-domain underlying the analysis.
- We propose an alternative definition of embedding of 3-valued logical structures, which allows abstract elements representing any number of concrete heap elements to represent no concrete nodes at all, yet still retaining the connectivity between other elements of the structure in a conservative manner. This redefinition instantly extends to the definition of the sub-domain of 3-valued canonical structures that is used during the analysis, as well as the definition of join and meet operators. An additional extension to the semantics of first-order logical quantifiers is required in order to assure soundness of the revised framework.
- We describe an implementation of the above extensions, as they were incorporated into the 3-valued shape analysis framework of [1]. Revising the abstract domain is achieved by applying a relatively small set of changes to its implementing code, thanks to the extendable nature of our design and the flexibility of our core algorithm for detecting relationships between abstract heap elements.
- We provide initial experimental results supporting the usability of the newly introduced abstraction. In particular, we demonstrate an up to 60% total state-space deflation, and up to 55% cut in analysis time, for some small yet non-trivial benchmark programs.

The rest of this paper is organized as follows. Section 2 introduces 3-valued logic based abstraction for conducting heap shape analysis, as well as surveys our previous work on a restricted variant of this framework. Section 3 explains the problem that is apparent with the current framework, and described our approach for addressing it by redefining the embedding relation between abstract heap structures. Section 4 discusses implementation of these extensions in our experimental shape analyzer. Initial empirical results are given in Section 5. Section 6 discusses related work, and Section 7 concludes.

## 2 3-Valued Logic Shape Analysis

We explain the representation of concrete program heap states and their abstraction, as well as the representation and manifestation of abstract transformers, based on the parametric framework of [5]. Then, we outline the approach used in recent work [1] to implement a restricted, yet highly effective 3-valued logic shape analyzer.

### 2.1 Abstract domain

#### 2.1.1 Concrete program states

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

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

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

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

Table 1 shows the predicates used to record properties of individuals for the analysis of the program in Figure 1. We also define additional so-called “instrumentation” predicates to capture properties of individuals such as pointer-aliasing, sharing, cyclicity, and transitive reachability. Instrumentation predicates provide for more precise information when applying abstraction on a concrete semantics. In particular, in Table 1 we define instrumentation predicates that capture reachability information (via predicates

Predicates	Intended Meaning
$eq(v_1, v_2)$	$v_1$ equals $v_2$
$\{x(v) : x \in PVar\}$	Variable $x$ points to object $v$
$n(v_1, v_2)$	The $n$ field of $v_1$ points to $v_2$
$\{r_{x,n}(v) : x \in PVar\}$	$v$ is reachable from variable $x$ along a sequence of $n$ fields
$s_n(v)$	Several $n$ fields point to $v$
$c_n(v)$	$v$ resides on a directed cycle of $n$ fields

Table 1: Predicates used for shape analysis of the program in Figure 1, and their intended meaning. The set of pointer variables in a program is denoted by  $PVar$

of the form  $r_{x,n}(v)$ ), sharing information (via the predicate  $s_n(v)$ ) and information on cycles in the heap graph (via the predicate  $c_n(v)$ ).

Concrete states (i.e., 2-valued logical structures) are depicted as directed graphs. Each individual of the universe is drawn as a node. A unary predicate  $p(u)$ , which holds for an individual  $u$ , appears next to the corresponding node. A unary predicate representing a reference variable is shown by having an arrow drawn from its name to the node referenced by the variable. The binary predicate  $n(u_1, u_2)$ , which holds for a pair of individuals  $u_1$  and  $u_2$ , is drawn as a directed edge from  $u_1$  to  $u_2$ , and labeled  $n$ . The predicate  $eq$  is not drawn, since any two nodes are different and every node is equal to itself.

Figure 2(a) shows a concrete program state arising after the execution of the statement  $t = y.n$  at the second loop of the program in Figure 1.

#### 2.1.2 Abstract program states

The abstract program states we use are based on Kleene 3-valued logic [5], 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  $\sqcup$ , defined by  $x \sqcup y = x$  if  $x = y$ , and  $x \sqcup y = \frac{1}{2}$  otherwise.

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

An abstract state may include *summary nodes*. A summary node is an individual which corresponds to one or more individuals in a concrete state represented by that

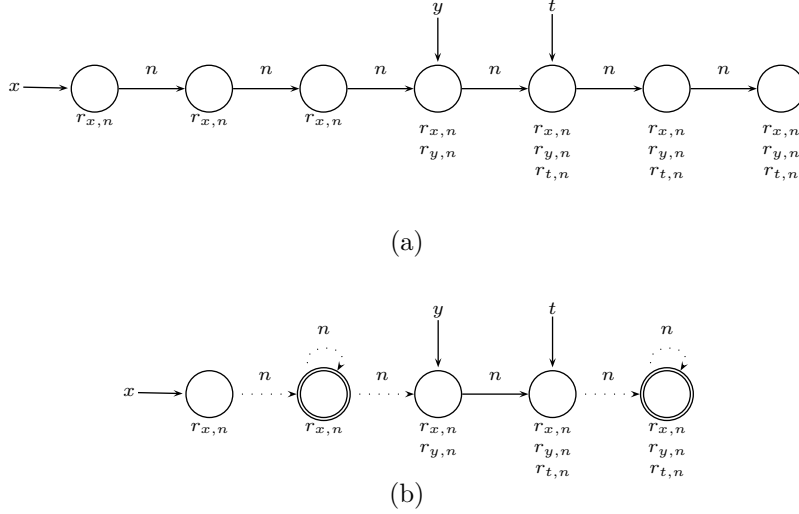


Figure 2: (a) A concrete program state arising after the execution of the statement  $t = y.n$  inside the second loop of the program in Figure 1; (b) An abstract program state approximating the concrete state in (a)

abstract state. A summary node  $u$  has  $eq(u, u) = \frac{1}{2}$ , indicating that it may represent more than a single individual.

Abstract (3-valued) logical structures are also depicted as directed graphs, where unary predicates denoting reference variables, 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  $3\text{-STRUCT}[\mathcal{P}]$ , and usually abbreviate it to  $3\text{-STRUCT}$ .

We define a partial order on structures based on the concept of *embedding*.

**Definition 3 (Embedding).** Let  $S = (U, \iota)$  and  $S' = (U', \iota')$  be two structures and let  $f : U \rightarrow U'$  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  $u_1, \dots, u_k \in U$ ,

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

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

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

**Definition 4 (Powerset order).** For sets of structures  $XS_1, XS_2 \subseteq 3\text{-STRUCT}$ ,  $XS_1 \sqsubseteq XS_2$  if and only if  $\forall S_1 \in XS_1 : \exists S_2 \in XS_2 : S_1 \sqsubseteq S_2$ .

In the following definition, we restrict sets of 3-valued structures by disallowing non-maximal structures, This ensures that the Hoare ordering is a proper partial order on powerset elements. We are now ready to present the general abstract domain used in our abstract interpretation framework.

**Definition 5 (Core abstract domain).** The abstract domain  $D_{3\text{-STRUCT}}$  consists of all finite sets of 3-valued structures that do not contain non-maximal structures,  $\{XS \subseteq 3\text{-STRUCT} \mid \forall S_1, S_2 \in XS : S_1 \sqsubseteq S_2 \implies S_1 = S_2\}$ , partially ordered as in Definition 3. Together with the obvious definition of a bottom element (the empty set), a top element (a set containing one empty structure, and one structure that contains a single summary node and whose predicates are all interpreted as  $\frac{1}{2}$ , for every node-tuple assignment), a join operator (set union minus non-maximal structures), and a meet operator (defined via join),  $(D_{3\text{-STRUCT}}, \perp, \top, \sqcup, \sqcap)$  is a lattice.

### 2.1.3 Bounded program states

Note that the size of a 3-valued structure is potentially unbounded and that  $3\text{-STRUCT}$  is infinite. This also implies that sets of such unbounded structures could potentially contain an infinite number of elements. The analyses studied in [5], one of which is implemented in [1], rely on a fundamental abstraction function for converting a potentially unbounded structure—either 2-valued or 3-valued—into a bounded 3-valued structure.

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

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

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

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

The abstraction function  $\beta_{\text{blur}}$  is called *canonical abstraction*, and serves as the basis for the abstract interpretation described in [5]. In particular, it serves as the basis for defining various different abstractions for the (potentially unbounded) set of 2-valued logical structures that may arise at a program point. We also define the function  $\alpha$ , which extends  $\beta_{\text{blur}}$  for sets of structures by  $\alpha(XS) = \bigsqcup\{\beta_{\text{blur}}(S) \mid S \in XS\}$ .<sup>2</sup> Along with a respective concretization function  $\gamma$ , that is defined by means of  $\alpha$ , the concrete domain  $D_{2\text{-STRUCT}}$  and the abstract domain  $D_{\text{B-STRUCT}}$  form a Galois connection, and hence comply with the requirements of [3] for constructing abstract interpretation frameworks.

## 2.2 Abstract transformers

The abstract interpretation framework described in [5] models the semantics of program transformations w.r.t. state predicates by first order logic formulas with transitive closure over the interpretation associated with each inbound structure. For example, the transformation induced by the program statement  $t = y.n$  from the example

in Figure 1, is modeled by

$$\begin{aligned} t(v) &\leftarrow \exists v' : y(v') \wedge n(v', v) \\ r_{t,n}(v) &\leftarrow r_{y,n}(v) \wedge (c_n(v) \vee \neg y(v)) . \end{aligned}$$

While the correctness of this transformer is evident when applied to some concrete structure, it is not at all obvious that the same holds for the case of an abstract structure, such whose predicates may evaluate to  $\frac{1}{2}$ . Nonetheless, the *embedding theorem* proved in [5] shows that the result of such a transformation is always a *sound approximation* of the best transformer.<sup>3</sup>

Although evaluating update formulas over abstract state elements in a straightforward manner is indeed sound, it would cause a significant loss of precision upon each transformation, thus leading to a useless analysis. In order to address this problem, and since a best transformer is generally intractable in this case,<sup>4</sup> two auxiliary algorithms are described in [5] in order to apply *partial concretization* of abstract structures.

**Focus.** Given a 3-valued structure  $s$  and some first order formula  $\phi$ , this algorithm essentially enumerates the set of structures which is embedded in  $\{s\}$ , and whose corresponding concrete range is equal to that of  $\{s\}$ , such that  $F$  evaluates to either 0 or 1, for each of its structures. Note, however, that the result of such enumeration might yield structures that do not satisfy the integrity constraints, as well as structures that are not precise enough with respect to instrumentation values associated with their individuals.

**Coerce.** Given a 3-valued structure, the functionality of this operator is two-fold: by exhaustively evaluating formulas derived from structure integrity rules, over all tuples of nodes, it both dismisses structures for which some constraint is breached, and also tightens predicate values where such a tightening is implied by the constraints. By that, the coerce step is normally applied following each focus step, so as to complement the weaknesses of the latter.

For brevity, we omit actual examples of the above operations, as well as details concerning the algorithms used to compute both. These can be found in [5].

<sup>1</sup>The notion of a bounded structure can be generalized by considering any subset of the set of unary predicates to serve as *abstraction predicates*, as done in TVLA. This generalization, however, is avoided in our restricted framework [1].

<sup>2</sup>The operator  $\bigsqcup$  is the least upper bound on the lattice  $D_{\text{B-STRUCT}}$ .

<sup>3</sup>Note, that the notion of a “best transformer” in the case of 3-valued abstract structures implies an instrumented concretization function, such that assures the validity of concrete structures w.r.t. the *integrity constraints* expressed by instrumentation predicates.

<sup>4</sup>Recent work has been done in [6] to elevate the applicability of best transformers for the case of 3-valued analysis, using a theorem prover to evaluate first order logic formulas.

## 2.3 Lightweight 3VL shape analyzer

In recent work [1] we proposed a specialized variant for a 3-valued logic shape analyzer, which deploys the above abstraction to effectively analyze intermediate representations of program in the form of control-flow graphs, that can in turn be automatically derived from Java programs. This framework is restricted by several means, compared to the reference implementation of [4].

**Fixed predicate arity.** Our framework is restricted by construction to support nullary, unary, and binary predicates only. While this is conceptually unnecessary and potentially prohibiting future extensions to the supported abstraction, we found that higher arity predicates are rarely used in practice. Furthermore, by intently omitting them we were able to use a graph-based data structure for representing logical structures, a concept that naturally lends itself to further topology oriented algorithms in our framework (e.g., computing transitive closure).

**Fixed abstraction instrumentation.** We support a predefined set of instrumentation predicates, which captures shape properties of any generic (recursive) data structure. It includes the predicates described in Table 1—that is, *reachability*, *cyclicity*, and *sharing*—as well as a *cancel* predicate  $c_{f_1, f_2}$ , which is instantiated over all pairs of binary predicates  $f_1$  and  $f_2$ , such that  $c_{f_1, f_2}(v)$  holds iff  $f_1(v, v')$  implies  $f_2(v', v)$ , for all  $v'$ .

**Fixed set of hard-coded transformers.** We support a universal set of intermediate level operations, including manipulation of reference and Boolean variables and object fields, branching based on Boolean conditions, memory allocation and deallocation, and procedure call and return (context sensitivity is not supported currently, though). By that, we are able to encode a variety of real-world Java programs.<sup>5</sup>

In the following, we will assume the above restrictive variations to be the baseline, on top of which we propose and examine further improvements. We will note specifically when such an extension relies on either of the above assumptions, and discuss its implications with respect to the general framework.

The following section describes a particular problem which is evident with our current analysis framework. We then argue that it can be improved via an alternative—and a more permissive—definition of embedding of abstract

<sup>5</sup>Note that support for array objects is currently not supported, and is considered to be future work.

structures, and explain how this choice affects the various parts of the abstraction.

## 3 Loose Embedding

Consider the example code in Figure 1. Analyzing the program using the abstract interpretation framework described in §2 yields a total of 109 heap abstractions, distributed among the 33 CFG nodes of the intermediate representation of the program. This implies an average of over 3 abstract heap descriptors per CFG node, with highest number being 16 structures per single node. What is causing that large number of distinct abstract states to occur? In the following, we attempt to highlight one source for such inflation.

### 3.1 State-space inflation in loops

Figure 3 shows three of the abstract structures representing distinct sets of concrete states of the heap, immediately past the statement  $t = y.n$  in the second loop traversing the linked list. (For now, ignore the arrows pointing from the structures in Figure 3(a) and Figure 3(c) to the structure in Figure 3(b).) The structure in Figure 3(b) represent a set of concrete linked lists, for which reference variable  $x$  points to the head element, followed by a sequence of (one or more) elements, followed by a pair of elements pointed by  $y$  and  $t$ , respectively, then followed by a sequence of (one or more) elements forming the list tail. Indeed, this structure describes a *general case* that the program exhibits while executing the traversal loop, and to that extent it contributes to the understanding of the behavior of the program, in addition to providing a conservative (sound) approximation on the set of concrete states incurred by the program at this point.

On the other hand, the other two structures in Figure 3 describe what could be considered a slight variant of the mentioned general case. Specifically, Figure 3(a) represent a set of lists that lack the suffix past the element pointed by  $t$ , and Figure 3(c) represents a set of lists that lack the infix in between the head element and the element pointed by  $y$ . A fourth structure arising at the same program point—which is not depicted in the figure—represents a list with neither infix or suffix nodes.

As is evident in this case, the actual number of state descriptors used to represent *special cases* is—together with the general case—exponential by the number of summary nodes contained in this general case. These special case descriptors are inevitable by construction of the abstraction framework, given that the loop traverses all elements of the list. Yet, informally speaking, they seem to contribute

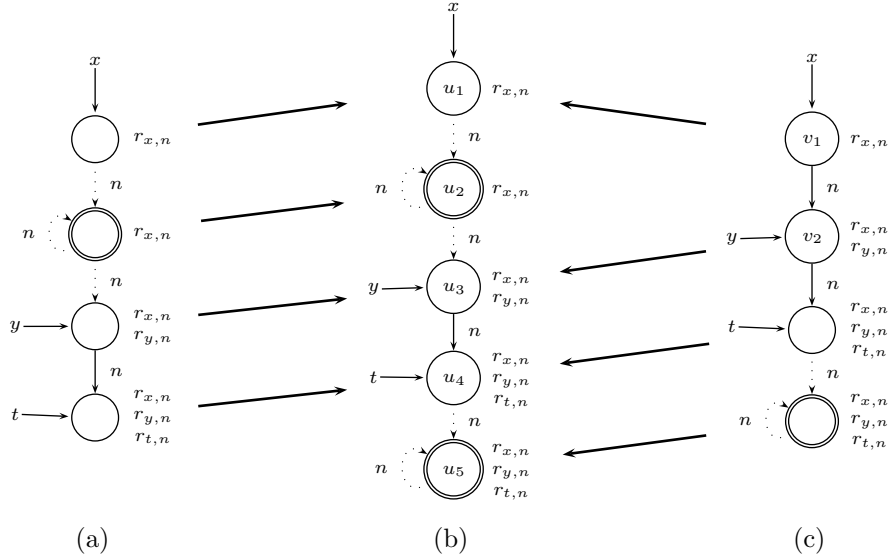


Figure 3: Three abstract heap descriptors arising after the statement  $t = y.n$  in the second loop of Figure 1. Structure (b) represents a loop that has  $x$  pointing to its head, followed by one or more nodes, then two consecutive elements pointed by  $y$  and  $t$ , respectively, and finally one or more nodes forming the list suffix. Structures (a) and (c) represent similar cases, only they omit either the suffix or the infix nodes, respectively

very little information compared to what the general case already expresses, consequently adding to the analysis only little precision at a high cost.

### 3.2 Relaxing the embedding relation

Recall that Definition 5 relies on the use of embedding relation to eliminate non-maximal structures from abstract powerset elements, such that the resulting abstract elements are non-redundant. In attempt to embed the special cases in Figure 3 in the general case, we consider the following relaxations to summary node semantics.

**Allow summaries to represent zero nodes.** In doing so, we attempt to relax the requirement for an embedding function to be surjective, as long as only summary nodes can be excluded from its range. Revisiting the above mentioned example, we see that the individuals of the structure in Figure 3(a) can be mapped to some individuals of the structure in Figure 3(b), as indicated by the bold arrows mapping individuals of the former to those of the latter, such that only the suffix summary individual  $u_5$  is not in the image of this map. Yet, it is clear that the requirement for predicate interpretation consistency in Definition 3 is satisfied, for all predicates in node tuples. This relaxation, therefore, allows for the structure in Figure 3(b)

to embed the structure in Figure 3(a), making the latter non-maximal, hence disposable.

#### Retain connectivity via non-mapped summaries.

Consider the non-surjective mapping as is depicted by the bold arrows pointing from the individuals of the structure in Figure 3(c) to those of the structure in Figure 3(b): here, as opposed to the previous case, the fact that  $u_2$  is excluded from the range of the mapping function, breaks the connectivity of the structure in Figure 3(b) compared to that of the structure in Figure 3(c). In particular, we see that the the interpretation of  $n(v_1, v_2)$  in the right-hand side structure evaluates to 1, whereas the interpretation for the image individuals  $n(u_1, u_3)$  evaluates to 0. Thus, this proposed function does not embed the structure of Figure 3(c) in that of Figure 3(b).

We therefore consider the following additional relaxation: allow predicate interpretation consistency of any binary predicate to be checked against the *constrained transitive closure* of that predicate, such that is only computed via summaries that are excluded from the range of the (candidate) embedding function. Here, since  $n(u_1, u_2) \wedge n(u_2, u_3) = \frac{1}{2}$ , we have that the interpretation consistency requirement is sat-

ified, hence the non-surjective mapping in the diagram can be considered to embed the structure of Figure 3(c) in that of Figure 3(b), thus making the former disposable.

We are now ready to give the formal definition of the alternative embedding relation suggested above. Note that we restrict this definition to predicates of arity at most two (i.e., binary predicates), as our variant of the 3-valued logic shape analyzer does not support predicates of higher arity whatsoever.<sup>6</sup>

**Definition 6 (Loose embedding).** Let  $S = (U, \iota)$  and  $S' = (U', \iota')$  be two structures and let  $f : U \rightarrow U'$  be a function, such that for every node  $v \in V = U' \setminus \{f(u) \mid u \in U\}$ ,  $eq(v, v) = \frac{1}{2}$ . We say that  $f$  *loosely embeds*  $S$  in  $S'$ , denoted  $S \sqsubseteq^f S'$ , if (1) holds for all nullary and unary predicates, and for every predicate  $p \in \mathcal{P}^{(2)}$  and a pair of individuals  $u_1, u_2 \in U$ ,

$$p^S(u_1, u_2) \sqsubseteq \bigvee_{v_1, \dots, v_k \in V} p^{S'}(f(u_1), v_1) \wedge \left( \bigwedge_{1 \leq i \leq k-1} p^{S'}(v_i, v_{i+1}) \right) \wedge p^{S'}(v_k, f(u_2)) .$$

Note that this formulation assumes the conjunction of the empty set to evaluate to 1. We say that  $S$  is *loosely embedded* in  $S'$ , denoted  $S \sqsubseteq S'$ , if there exists a function  $f$  such that  $S \sqsubseteq^f S'$ .

Note that the above definition immediately extends to the abstraction and its associated operators as given in Definition 5, as well as its derived bounded state subdomain.

### 3.3 Abstraction monotonicity

We start by formally stating the fact the loose embedding indeed *relaxes* traditional embedding, and induces a transitive relation, therefore implying the monotonicity of the induced abstraction function.

**Observation 7.** Let  $S, S' \in D_{3\text{-STRUCT}}$ , then  $S \sqsubseteq S'$  implies  $S \sqsubseteq^f S'$ .

This is an obvious fact, since an embedding function as per Definition 3 is a special case of an embedding function as per Definition 6, where the function is surjective, therefore all individuals of the target structure are contained in the function's range.

<sup>6</sup>Possible ways to generalize this notion for higher arity predicates may be considered, but those are omitted from this paper.

**Observation 8.** The loose embedding relation in Definition 6 is transitive.

This fact is also immediate by construction of the loose embedding relation, and due to the transitive nature of Kleene values ordering. Note, however, that loose embedding is not antisymmetric for cases of 3-valued structures in general (3-STRUCT), therefore it induces a partial *pre-order*, rather than a partial order.<sup>7</sup>

We now assume the core abstract domain  $D'_{3\text{-STRUCT}}$ , along with respective domain operators  $\tilde{\sqcup}$  and  $\tilde{\sqcap}$ , to be defined in an analogous manner to their definitions in §2.1.2 with the single difference that traditional embedding is substituted with loose embedding. Note, again, that since the relation underlying the set of abstract structures is a partial preorder, the values of join and meet are not necessarily unique (still they are well defined). Consequently, we assume that the bounded abstract domain  $D'_{B\text{-STRUCT}}$  and the abstraction function  $\alpha' : D_{2\text{-STRUCT}} \rightarrow D'_{B\text{-STRUCT}}$  are defined analogously.<sup>8</sup>

**Lemma 9.** *The abstraction function  $\alpha'$  is monotonic.*

*Proof.* Let  $XS_1, XS_2 \in D_{2\text{-STRUCT}}$ , such that  $XS_1 \subseteq XS_2$ . By definition of  $\alpha$  and  $\alpha'$ , and by Observation 7, we have that, for every  $XS \in D_{2\text{-STRUCT}}$ ,

$$\begin{aligned} \alpha'(XS) &= \tilde{\bigsqcup} \{ \beta_{blur}(S) \mid S \in XS \} \\ &= \tilde{\bigsqcup} \{ S' \mid S' \in \alpha(XS) \} , \end{aligned} \tag{2}$$

implying  $\alpha'(XS) \subseteq \alpha(XS)$ .

Let  $S_1 \in \alpha'(XS_1)$ . Then  $S_1 \in \alpha(XS_1)$ , therefore by monotonicity of  $\alpha$  and by Definition 4, there exists  $S_2 \in \alpha(XS_2)$  such that  $S_1 \sqsubseteq S_2$ , implying  $S_1 \sqsubseteq S_2$ . If  $S_2 \in \alpha'(XS_2)$ , then there is nothing to show. Otherwise, by the (implicit) definitions of  $\alpha'$  and loose join, there exists some  $S'_2 \in \alpha'(XS_2)$  such that  $S_2 \sqsubseteq S'_2$ . Thus, by transitivity of loose embedding we have  $S_1 \sqsubseteq S'_2$ . It follows that, by Definition 4,  $\alpha'(XS_1) \sqsubseteq \alpha'(XS_2)$ .  $\square$

### 3.4 Soundness

As stated in §2.2, the soundness of transformers expressed as first-order logic formulas is guaranteed thanks to the

<sup>7</sup>... the full consequences of which are not yet fully clarified.

<sup>8</sup>There is a certain subtlety I did not address here: while loose embedding is not antisymmetric for the case of general 3-valued structures, it is antisymmetric for the case of canonically bounded structures, as defined in §2.1.3. Therefore, the result of the loose join operator in the definition of  $\alpha'$  is unique, implying that  $\alpha'$  itself is indeed a well-defined function, hence so is  $\gamma'$ . Informally speaking, proving antisymmetry here relies on the ordering of Kleene values, showing that any mutually loosely embedded structures must be isomorphic. The formal proof of this claim is omitted though.

embedding theorem. Nonetheless, given the new semantics we have associated with summary nodes, following which a summary node in some structure might represent no nodes in an embedded structure, we need to revise the semantics of logical quantifiers in order to account for this extension.

**Existential quantification.** We interpret each occurrence of the form  $\exists u.P$ , with  $P$  being some predicate, as  $\exists u.eq(u, u) \wedge P$ . This guarantees that any predicate that is existentially quantified over a summary node is “lowered” to  $\frac{1}{2}$ , accounting for the fact that it may not exist in some concrete setting.

**Universal quantification.** We interpret each occurrence of the form  $\forall u.P$  as  $\forall u. \neq eq(u, u) \vee P$ . This guarantees that any predicate which is universally quantified over (one or more) summaries, will be “raised” to  $\frac{1}{2}$ , thus accounting for possibly non-existent nodes in some concrete settings.

We argue (informally) that these changes suffice in order to retain the soundness of local transformers, and omit the formal proof for brevity.

**Theorem 10.** *The framework, resulting from the introduction of loose embedding to 3-valued shape analysis, yields a valid abstract interpretation, hence it is sound.*

We omit the formal proof of this theorem, and satisfy for stating that it follows from the above extensions to retain local soundness, as well as Lemma 9.

In the following section we describe the practical implications of extending our static analysis framework to support loose embedding, including the algorithmic extensions required in order to compute it.

## 4 Implementation

In §2.3 we give a brief outline of the restricted 3-valued shape analysis framework, whose implementation was detailed in [1]. This framework, which leans heavily on the effective implementation of join and meet operators, uses a core algorithm to reveal relationships between 3-valued structures, such as embedding (for computing join) and correspondence (for computing meet). The generality of this procedure is a key feature in extending our framework to support loose embedding abstraction.

### 4.1 Binary transitive closure

Underlying the test for loose embedding is the ability to form the transitive closure of binary predicates, between

any two nodes covered by the embedding function, and across all sequences of (summary) nodes not covered by it. This closure is computed on demand, before verifying embedding of any two structure, and given some mapping individuals of one structure to those of the other structure.

For brevity, we omit the tedious algorithmic details, and settle for stating the general concept of the computation. Given a candidate embedding structure, with proper indication of individuals excluded from the range of the tested embedding function, the algorithm initializes a work queue with the set of such nodes. Then, it iterates repeatedly, extracting one element from the queue at a time, and propagating (“raising”) Kleene values of binary predicate interpretation in a backward manner—that is, destinations to sources, from the topological point of view. Whenever an edge (predicate value) is updated, its source vertex (individual) is added to the work queue. The algorithm terminates when the queue is empty.

Since the above is an instance of a general purpose chaotic iterations algorithm, over a finite domain of Kleene values assign to a finite set of predicates and over a finite set of nodes (individuals), it is guaranteed to terminate. This fact also informally implies the correctness of the algorithm. The formal proofs for the above are omitted for brevity.

### 4.2 Extended domain operators

In adjusting the implementation of the join and meet operators—as proposed in [2]—to support loose embedding abstraction, we settle for outlining the required extensions and omit the details.

**Relaxed matching quota for summaries.** Since—conforming to the new embedding definition—summary nodes may represent zero nodes, we relax the lower matching quotas associated with summaries to zero. Thus, our matching procedure can also consider relations where summary nodes are not necessarily matched. For further details regarding the role of the matching procedure in the computation of domain operators, see [2].

**Applying transitive binary closure.** Each of the two domain operators applies the above described transitive closure procedure, although in a slightly different way. The computation of join requires that constrained transitive closure is applied to the target (embedding) structure, for each mapping returned by the matching procedure, prior to checking the consistency of predicate interpretation between the structures. This technique is derived from Definition 6 in

a straightforward fashion.

In the case of computing meet, transitive binary closure needs to be evaluated on both operand structures, before verifying consistency and deriving an intermediate structure, for each inferred matching relation. While a slightly more complicated procedure, it was still surprisingly simple to implement, thanks to the extendable nature of the matching procedure, and the method that is used to formulate the meet value. For further details, see [2].

### 4.3 Additional adjustments

Following §3.4, we extended the semantics of first-order logical quantifiers in the evaluation of update formulas, that form the abstract transformers in our framework. This includes the consideration of the *eq* predicate when enumerating existential and universal quantification, as described above.

One enhancement that has been made possible due to incorporation of loose embedding, is the pruning of refinement sets associated with abstract transformers. As explained in [1], our framework relies on the exclusive use of meet and join operators for performing partial structure concretization, and utilizes predefined sets of structures corresponding to the desired property to be “focused”. Switching to loose embedding abstraction allows us to omit structures that are consequently non-maximal from our refinement sets, leading to smaller sets, in turn promoting quicker application of transformers.

It is worthwhile stating that the relaxation of embedding does not affect the core canonical abstraction of single structures, and therefore revising the domain operators is sufficient to encompass the whole change to the framework abstraction.

All in all, the changes that needed to be applied to the existing framework were of quite a modest scope, essentially affecting three components in our system—abstract structure, abstract transformers and refinement implementations—and in a fairly mild way. We accredit this to the minimalist approach of the proposed extension, as well as the extendable nature of our framework.

## 5 Experimental Results

Due to the scope of the project, we were not able to conduct extensive evaluation of the loose embedding abstraction. Table 2 presents initial results applying our extended framework to a pair of micro-benchmark Java programs,

		sll-loop			sll-delete		
		$\sqsubseteq$	$\tilde{\sqsubseteq}$	$\Delta\%$	$\sqsubseteq$	$\tilde{\sqsubseteq}$	$\Delta\%$
# loops		2			3		
# locations		33			49		
# struct.	total	109	59	45	573	227	60
	ave	3.3	1.8		11.7	4.6	
	peak	9	4	55	69	22	68
time (ms)		22	20	9	513	227	55

Table 2: Benchmark results for a two Java programs processing singly-linked lists, using both strict- and loose-embedding abstractions. First two rows indicate the number of loops and total number of nodes in the CFG graph forming the intermediate representation of each program. Last row indicates total analysis time

both manipulating singly-linked lists by means of constructing a list of arbitrary length (namely, allocating heap objects and linking them), traversing it, and modifying its structure (thus incurring destructive update to reference fields).

One difference between the two programs, except for their apparent difference in size, is the fact that one of the loops in the second program—sll-delete—may terminate abruptly, such that the traversing pointers may be at any location throughout the list. This behavior is generally characterizing a search-oriented iteration, as opposed to whole-list traversal applied by other loops in our benchmarks. While there is nothing abnormal about a program fragment of this kind, it is of special interest to our problem, since it implies that a potentially bigger set of states can “escape” the loop cycle and get transformed and propagated through subsequent parts of the program. Indeed, this situation is evident in the high number of average structures per CFG node for this benchmark, as well as the peak number of structures per node (69), that a traditional abstraction yields, compared to the first benchmark.

The numbers shown in Table 2 call for several observations regarding the effectiveness of the loose embedding abstraction. First, it is apparent that a significant deflation of the fixed-point state-space—up to 60% in the case of sll-delete—is achieved thanks to the use loose embedding, and that the rate of this improvement increases as program size and complexity grow. This supports our initial conjecture that loops in the program induce a potentially large number of structure that can be considered redundant, in terms of their contribution to the precision and usability of the results. Second, we see that loose embedding is especially effective in flattening the state-space distribution, by

cutting the number of structures associated with the most congested program locations (specifically, these are nodes within one of the loops in each program, that exhibit the highest number of distinguishable states). This fact as well supports the previous claim, and establishes the effectiveness of loose embedding for highly congested nodes. Third, we see that analysis times drop significantly—namely, up to a 55% cut for the case of `sll-delete`—a fact which validates our expectation that the reduction in state-space expansion would pay off the computational costs induced by the incorporation of loose embedding in the analysis.

Finally, it is worth mentioning that the actual results of a loose embedding based analysis are by far more comprehensible—and therefore, more usable—compared to those of a traditional (strict) analysis. We consider this a nice practical outcome, which supports our claims concerning the problems with current shape abstraction.

## 6 Related Work

This work shares common goals with a few other efforts, all aiming at improving the scalability of shape analysis, hence increasing its usability for practical purposes. While some of these approaches deviate in part or in full from the definition of the abstract domain—namely, by suggesting alternative means to canonical abstraction that may yield more compact descriptors, or by considering alternative definitions for canonization—most of them are incomparable to this work, and incur various limitations on the framework (e.g., introducing potentially unbounded abstract lattices) or on the kind of programs that can be analyzed (e.g., predicate abstraction that is specialized for analyzing linked lists).

One approach that corresponds with ours was implemented in TVLA [4], and is generally an attempt to capture the evolving environment of an abstract state using a single structure. This approach, commonly referred to as *semi-active nodes*, uses an additional predicate *ac* to annotate nodes that “are”, “are not”, are “may be” active in various concrete structures that are represented by some abstract structure. This approach may be considered somewhat more general compared to ours, as it allows any node—and not necessarily *any* summary node—to be excluded from the range of an embedding function, and also makes this feature explicit by the interpretation of the *ac* predicate per each node. However, we also think it is inferior to our approach by several means. The two noticeable ones include the excess imprecision induced by allowing any node to be potentially excludable, yet accommodating all possible (conservative) interpretation of predicates at the same time. This is opposed to our approach, by which

transitive binary closure is only considered once an embedding mapping—along with its induced set of excluded summaries—was chosen. The other relative advantage of our approach, is the clean applicability it implies as per the actual implementation of the framework, namely the small number of extensions that is required to accommodate it. Note, however, that due to the scope of the project, we did not conduct any quantitative or qualitative comparison between the two approaches.

## 7 Conclusion

In this work, we described an effort to improve scalability of a 3-valued logic shape analysis framework, by means of deflating the state-space exhibited by the analysis. Observing that certain patterns in the program induce a large number of abstract states, we argued that a (potentially large) number of them are redundant in terms of precision and usability they add to the analysis results, and proposed an alternative definition to the embedding relation, which underlies the determination of non-maximality of abstract structures. We proved that our extensions lead to a sound abstraction, and described the algorithmic extensions that were required in order to accommodate it in our framework implementation. We demonstrated the effectiveness of our approach on a pair of micro-benchmark programs, and argued that the results suggest that the opportunity for improvement is correlated with the size and complexity of the program. Finally, we discussed related work.

We believe that the next step in this research has both theoretical and practical aspects: as for the former, it seems that a more detailed analysis of the implications of introducing the new relation—which does not induce a strict partial order—is in place, in order to establish the correctness of our abstraction, and argue the determinism of the analysis results. We could also use a more elaborate proof of soundness, and argue the correctness of the algorithms used to implement loose embedding. From the practical point of view, it is evident that a bigger set of (bigger) benchmarks needs to be carried out, in order to support our empirical results and initial conjectures. Hopefully, these will be addressed in the near future, by means of extending our framework to provide end-to-end analysis of high-level code.

## Acknowledgments

I’d like to thank Mooly Sagiv and Roman Manevich for their useful feedback, and continuous willingness to help.

## References

- [1] G. Arnold. A topology-based approach for lightweight 3-valued logic shape analysis. CS264 course project, EECS Department, UC Berkeley, May 2005.
- [2] G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Combining shape analyses by intersecting abstractions. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3855, pages 33–48. Springer-Verlag, 2006. To appear.
- [3] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL)*, pages 269–282. ACM Press, 1979.
- [4] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS)*, pages 280–301, 2000.
- [5] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.
- [6] G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 530–545, 2004.