

Specialized 3-Valued Logic Shape Analysis using Structure-Based Refinement and Loose Embedding

Gilad Arnold

University of California, Berkeley
arnold@cs.berkeley.edu

Abstract. We consider a shape analysis framework based on 3-valued logic, and explore ways for improving its performance and scalability by means of reducing algorithmic overhead and restraining abstract state set inflation. First we propose a new approach to implementing a fast 3-valued logic analyzer, which replaces a general-purpose abstract heap refinement mechanism—accounting for most of the time spent by the reference implementation—with tailored structure-based refinement. We apply our framework to analyze a set of small Java programs manipulating singly- and doubly-linked lists, obtaining results that are comparable to those of the reference implementation, with a process 40–85 times faster and 2–11 times less memory consuming. We then propose a new definition for partial ordering of abstract heap descriptors (embedding), that trims abstract states representing “special cases” in the presence of a state representing a “general case”. This extension deflates sets of abstract states by a combinatorial factor, resulting in 45–55% less structures for the same set of benchmarks. Despite its induced algorithmic overhead per operation, this modification further cuts the analysis time by 17–50%. We argue that improving on these two axes together yields a promise for greater applicability of specialized shape analysis to real-life programs.

1 Introduction

The ability to reason about the set of heap configurations that a computer program may exhibit, without actually running the program, has many uses in program analysis and verification. These include whole-program verification tasks, like verifying the absence of null dereferences; proving correctness of heap intensive programs, such as heap sorting algorithms [8]; and checking properties of heap references throughout the program, such as dead objects analysis and its applications to static garbage collection [3]. Nonetheless, shape analysis appears to be among the hardest problems in static program analysis: proving even simple properties of very small programs manipulating dynamically allocated data structures is generally undecidable, and even the compulsory deployment of conservative abstraction methods following Cousot & Cousot [4] implies non-trivial

frameworks, in turn inducing 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 the abstraction, transformations, and various domain operators.

We consider a shape analysis framework that models heap topology and related properties using logical structures and applies first-order logic formulas to model program semantics [12]. Although analyses instantiated by this framework give precise and meaningful results compared to actual (concrete) heap structures exhibited by a program, it is not widely studied, let alone deployed in actual production-level compilers or analysis tools. Indeed, the 3-Valued Logic Analyzer (TVLA) [9] reference implementation was used to demonstrate the analysis precision and adaptivity to a wide variety of shape-related problems. However, analyzing even tiny programs manipulating linked lists can take as long as seconds. Designed as an extensible analysis generator, TVLA is under-optimized compared to a (presumed) specialized implementation. Still, we can observe at least two aspects which make a reference implementation inherently expensive.

Costly refinement and validation of abstract heaps. A significant portion of the analysis time—namely, up to 90%—is due to particular algorithms that are being used for refining abstract heaps.

State set inflation. The huge abstract domain underlying the analysis—whose induced complexity is doubly-exponential in the number of abstraction predicates (essentially, the number of reference variables in the program)—leads very quickly to a blow-up in the number of heap states being tracked by the analysis, even for mildly complicated programs.

```
x = null;
while (...) {
    y = new DLL();
    ...
    if (x != null) x.p = y;
    y.n = x;
    x = y;
}

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

Fig. 1. A Java program that constructs and traverses a doubly-linked list.

Fig. 1 shows a simple program that constructs and traverses a doubly-linked list. Analyzing an automatically generated dataflow representation of it using the default shape abstraction for linked lists [9] yields a total of 113 abstract heap structures and takes 1.4 seconds to complete with stock TVLA. A slightly more complicated example—a program that manipulates a singly-linked list using three loops, one of which removes an arbitrary element from the list—results in a total of 485 abstract heap descriptors and takes as long as 12 seconds to complete. This demonstrates the steep abstract states inflation and the respective time penalty experienced with programs of increasing complexity.

This paper describes the fresh implementation of a 3-valued logic based shape analysis tool. It is intentionally restricted compared to the fully-parameterized reference implementation, but appears to be better suited for performance and scalability, making the following major contributions.

Specialized structure-based refinement. While using a meet operator for abstraction refinement has already been suggested [3] it was never put into practice with the 3-valued logic framework. We take this concept to the extreme, performing merely all refinement tasks using a sequence of meet and join operations with sets of predefined structures, as explained in Section 3. Consequently, we are able to produce results that are as precise as those achieved using more powerful algorithms, in only a fraction of the time.

Loose embedding. We identify a case for overly elaborate abstract states that neither contribute to precision nor bear a significant descriptive insight as to the represented set of concrete states. Consequently, we propose an alternate definition of embedding of 3-valued logical structures, which allows abstract elements representing one or more concrete heap elements to represent no elements at all, yet still retains connectivity between other elements of the structure in a conservative manner. This extension—explained in Section 4—instantly constrains the abstract domain, and therefore the set of abstract states explored during the analysis. With proper further adjustments to the semantics of abstract transformers, we are able to restate the soundness of the framework.

Implementation and preliminary results. We have implemented the above techniques in our new shape analyzer and applied it to a small set of interesting micro-benchmarks as described in Section 5, showing an overall speed-up of up to 124 and an up to 15 times smaller memory footprint.

2 Basics of 3-Valued Logic Shape Analysis

We explain the heap state abstraction and abstract transformers following Sagiv *et al.* [12], then state the restrictions assumed as the baseline for our specialized analysis.

2.1 Concrete Program States

We represent concrete program states using 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 structure and ι is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k , $\iota(p) : U^k \rightarrow \{0, 1\}$.

Predicate	Intended meaning
$eq(v_1, v_2)$	v_1 equals v_2
$p(v)$	Variable p points to object v
$f(v_1, v_2)$	The f field of v_1 points to v_2
$r_{p,f}(v)$	v is reachable from variable p along a sequence of f fields
$s_f(v)$	Several f fields point to v
$c_f(v)$	v resides on a directed cycle of f fields references
$b_{f_1, f_2}(v)$	The f_2 field of an object pointed by the f_1 field of v points back at v

Table 1. Predicates used in the analysis of programs manipulating doubly-linked lists, with p (f) instantiated over the set of reference variables (fields).

Table 1 shows the set of predicates used in the analysis of the program in Fig. 1, with p and f instantiated over $\{x, y, t\}$ and $\{n, p\}$, respectively.¹ We require that the set of predicates includes the binary predicate eq , bearing the semantics of *equality* between individuals. Note the use of *instrumentation predicates*—like transitive reachability, shared referencing, cyclicity, and back-pointing—in addition to core shape predicates, the importance of which in retaining abstraction precision has been widely discussed [9,12].

A concrete state is depicted as a directed graph, where each individual in the universe is a node. The set of unary predicates that hold for each node appear right next to it. A unary predicate representing a reference variable that points to some node v is depicted by an arrow from the variable’s name to v . A binary predicate f which holds for a pair of individuals v_1 and v_2 is depicted by an f -labeled directed edge from v_1 to v_2 . The predicate eq is not shown, since any two nodes are different and every node is equal to itself.

Fig. 2(a) shows a concrete program state arising after the execution of the statement $\mathbf{t} = \mathbf{y.n}$ at the second loop of the program in Fig. 1. We denote the set of all 2-valued logical structures over a set of predicates \mathcal{P} by $2\text{-STRUCT}[\mathcal{P}]$, abbreviated to 2-STRUCT under the simplifying assumption that \mathcal{P} is fixed.

2.2 Abstract Program States

We represent abstract program states using Kleene 3-valued logic [12], an extension of Boolean logic which introduces a third value $\frac{1}{2}$ denoting a truth value that may be either 0 or 1. We utilize the partial order defined by $0 \sqsubseteq \frac{1}{2}$ and $1 \sqsubseteq \frac{1}{2}$, with the join operation defined accordingly.

¹ Note that b_{f_1, f_2} is only instantiated for pairs of *distinct* reference fields.

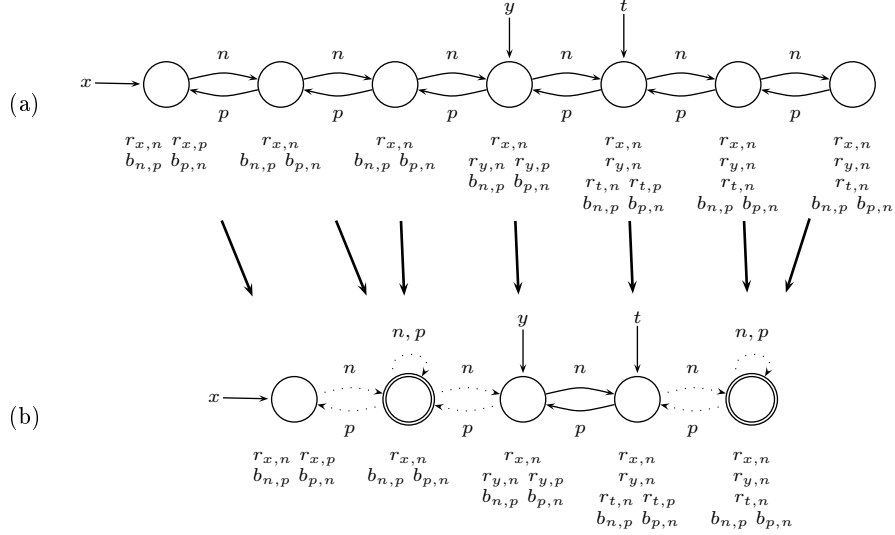


Fig. 2. (a) A concrete program state arising after the execution of the statement $t = y.n$ in Fig. 1; (b) An abstract program state approximating (a).

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 structure and ι is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k , $\iota(p) : U^k \rightarrow \{0, 1, \frac{1}{2}\}$. A summary node in an abstract state is an individual u for which $eq(u, u) = \frac{1}{2}$, representing one or more concrete nodes.

An abstract state is also depicted as a directed graph, where parenthesized predicates (unaries) and dotted arrows (binaries) denote $\frac{1}{2}$ values, and summary nodes appear as doubly-lined nodes. Fig. 2(b) shows an abstract state with two summary nodes, representing any number of one or more concrete nodes at the infix and suffix of the list, respectively.

We denote the set of all 3-valued logical structures over a set of predicates \mathcal{P} by 3-STRUCT[\mathcal{P}], abbreviated to 3-STRUCT. Note that Definition 2 generalizes Definition 1, therefore 2-STRUCT \subseteq 3-STRUCT.

We define a partial order on structures based on the concept of *embedding*, and extend it to a preorder on *sets* of structures.

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

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

S is embedded in S' , denoted $S \sqsubseteq S'$, if there exists f such that $S \sqsubseteq^f S'$.

The concrete structure in Fig. 2(a) is embedded in the abstract structure in Fig. 2(b) with respect to the mapping depicted by the bold arrows.

Definition 4 (Powerset embedding). *Given two sets of structures $XS, XS' \subseteq 3\text{-STRUCT}$, $XS \sqsubseteq XS'$ iff for all $S \in XS$ there exists $S' \in XS'$ such that $S \sqsubseteq S'$.*

In the following, we restrict sets of 3-valued structures by disallowing non-maximal structures. This ensures that the above Hoare order is indeed a proper partial order. The set $D_3\text{-STRUCT}$, consisting of all finite sets of 3-valued structures that do not contain non-maximal structures, along with the partial order given by Definition 4, form the *abstract domain* underlying our framework. We use the same order to define the concretization of a set of 3-valued structures, given by $\gamma(XS) = \bigcup_{XS' \sqsubseteq XS} XS'$.

2.3 Bounded Program States

Note that the size of a 3-valued structure is potentially unbounded. Therefore, 3-STRUCT contains sets with an infinite number of structures and is in turn infinite. We use a fundamental abstraction method [12] to convert a state descriptor of any size into a bounded (abstract) one.

A 3-valued structure $S = (U, \iota)$ is said to be *bounded* if for every two distinct individuals $u_1, u_2 \in U$, there exists a unary predicate p such that $p^S(u_1)$ and $p^S(u_2)$ evaluate to distinct *definite* truth values (*i.e.*, 0 and 1). The abstract domain $D_{\text{B-STRUCT}}$ is a finite sub-lattice of $D_3\text{-STRUCT}$, containing all (finite) sets of bounded structures that do not contain non-maximal structures. The structure abstraction function β —referred to as a *canonical abstraction* [12]—maps a potentially unbounded 2-valued structure into a bounded 3-valued structure. Namely, $\beta((U, \iota)) = (U', \iota')$, where U' consists of the disjoint subsets of U in which no unary predicate evaluates to distinct definite values, and for any pair of individuals in different subsets, there is at least one predicate which evaluates to distinct definite values. The interpretation ι' of each $p \in \mathcal{P}^{(k)}$ and k individuals $c_1, \dots, c_k \in U'$ is given by

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

Fig. 2(b) shows the bounded structure obtained from Fig. 2(a) (note that $S \sqsubseteq \beta(S)$ for all S). Powerset abstraction is given by $\alpha(XS) = \bigsqcup_{S \in XS} \{\beta(S)\}$.²

2.4 Abstract Semantics

The abstract interpretation framework of [12] models the semantics of program transformations using first order logic formulas with transitive closure. For example, the update to the value of the unary predicate t through program statement $\mathbf{t} = \mathbf{y.n}$ from the example in Fig. 1 is modeled by $t(v) \leftarrow \exists v' : y(v') \wedge n(v', v)$.

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

The *embedding theorem* [12] ensures that the result of a transformation on any abstract state is a sound approximation of the best transformer. Yet, as straightforward evaluation of update formulas over bounded abstract states leads to considerable loss of precision, and since a best transformer is generally intractable, we achieve *partial concretization* (*i.e.*, refinement) by means of two auxiliary operations [12].

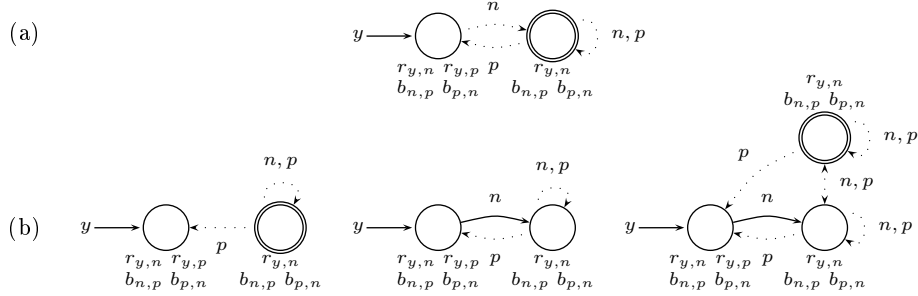


Fig. 3. Structure refinement using focus: (a) the original inbound structure; (b) structures resulting from focus using the formula $\exists v'.y(v') \wedge n(v', v)$.

The *focus* operation applies semantic reduction to a given 3-valued structure such that the evaluation of the first-order logic focus formula on any resulting structures yields a definite truth value (*i.e.*, 0 or 1). Fig. 3(a) shows a canonically bounded doubly-linked list structure that is being focused—prior to an update due to $\mathbf{t} = \mathbf{y} \cdot \mathbf{n}$ —using the formula $\exists v'.y(v') \wedge n(v', v)$. The resulting structures are shown in Fig. 3(b). However, note that focus might lead to structures that do not necessarily satisfy the integrity constraints—such as the leftmost structure in Fig. 3(b)—or are not as precise as could be with respect to the values of instrumentation predicates, such as the middle structure in Fig. 3(b). The functionality of the *coerce* operator in this regard is two-fold: by exhaustively evaluating formulas derived from structure integrity rules, it both dismisses structures for which some constraint is breached and also tightens predicate values where such a tightening is accommodated by the constraints. Thus, a *coerce* step normally follows a *focus* operation, so as to complement the weaknesses of the latter.

It should be noted that this refinement method—in particular the *coerce* step—is by far the most time consuming phase of the analysis in practice, suggesting that an alternate approach may be highly beneficial for efficiency.

2.5 Restricted 3-Valued Logic Shape Analysis

In this work, we assume a restricted instance of the parametric 3-valued logic framework [12] to be the baseline for further proposed improvements. Limiting predicate arity to nullary, unary, and binary predicates only, we further restrict

ourselves to the predefined fixed set of instrumentation predicates described in Table 1, which can capture shape properties of any (recursive) data structure. Finally, we support a fixed, universal set of intermediate-level operations—including manipulation of reference and Boolean variables/fields and basic control statements—which allows us to encode a variety of real-world Java programs.³

3 Specialized Analysis with Structure-Based Refinement

We describe in [1] the design and implementation of a specialized prototype shape analysis tool, constructed with the long-term goal of exploring techniques for faster practical shape analysis. Despite its conformance with the guidelines in Section 2.5, we believe that only one of these restrictions—namely, limited predicate arity—is inherent to the design, bearing the least actual burden to the applicability of derived analyses. Extending the framework beyond the aforementioned limitations is considered future work.

One interesting aspect of our implementation is the heavy use it makes of fine-tuned domain operators (*i.e.*, join and meet). Although both are hard problems given the domain of unbounded 3-valued structures,⁴ by re-using algorithms developed by Arnold *et al.* [3], we are able to infer certain relationships between 3-valued structures—such as deciding an embedding relation—in a surprisingly effective manner. This feature is heavily relied upon when we introduce structure-based, operator-intensive refinement. Also, we are able to exploit the flexibility of a fundamental graph matching technique as we later loosen the definition of structure embedding. For further details, the interested reader is referred to [1].

The fact that a meet operator can be used to perform abstraction refinement—both focusing an abstract structure prior to an update, as well as filtering structures based on some semantic condition—has already been discussed elsewhere [2]. Conforming to this approach, a 3-valued structure is used to express the desired semantic condition, and a meet operation is used to extract the subset of structures that are both represented by a given abstract state *and* comply with the semantic condition expressed by the refining structure. We have taken this approach to the extreme, essentially doing all abstraction refinements using meet (and join) operations. We demonstrate this approach by describing a simplified structure-based refinement operator for the abstract transformer of $\mathbf{t} = \mathbf{y}.\mathbf{n}$ from Fig. 1. This refinement operator—requiring that the \mathbf{n} field of the object pointed by \mathbf{y} is focused—is by far the most complicated one, mainly due to the subtle sub-cases that need to be considered for obtaining sufficient precision.

3.1 Sufficiently Tight and Effective Refinement

For simplicity, we assume that any semantically sane input structure is such that its y predicate evaluates to 1 for exactly one individual (*i.e.*, the dereference $\mathbf{y}.\mathbf{n}$

³ Array objects and call/return context-sensitivity are currently not supported.

⁴ The meet operator was shown particularly hard, even for bounded structures [3].

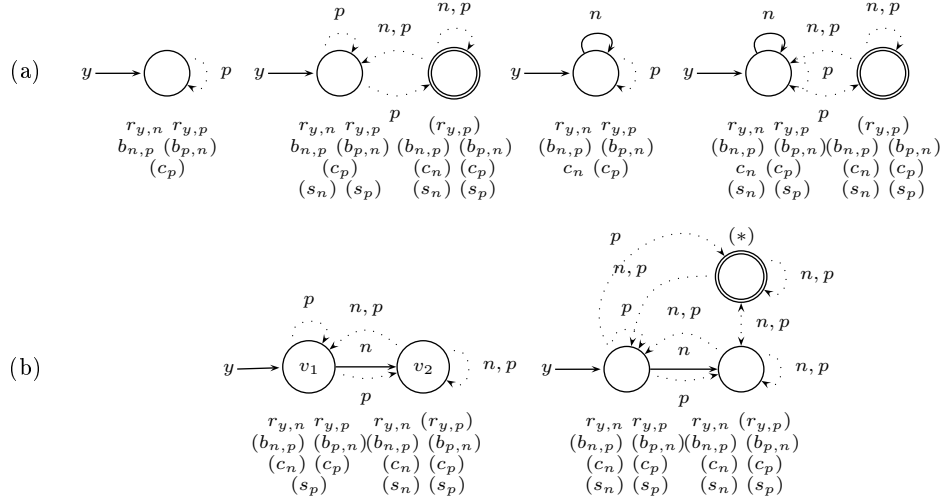


Fig. 4. A simplified structure-based refining set for $\mathbf{t} = \mathbf{y.n}$: (a) cases of null or self-loop $\mathbf{y.n}$; (b) cases of $\mathbf{y.n}$ pointing to a different node.

deterministically succeeds).⁵ Therefore, for the purpose of refinement, we can initially use a set of structures consisting of the three distinct cases where $\mathbf{y.n}$ is either (a) null, (b) a self-loop, or (c) points to a different node. Note that any of these general cases needs to be split into disjoint sub-cases, indicating whether additional nodes—other than the one pointed to by \mathbf{y} and (possibly by) $\mathbf{y.n}$ —may exist. Such a refinement set, with the simplification of ignoring x , t , and their induced instrumentation predicates, is shown in Fig. 4.

In this example, the refining structures impose very few constraints on the values of binary predicates between the different nodes—other than focusing the \mathbf{n} field of the object pointed to by \mathbf{y} —and, consequently, on those of instrumentation predicates. Still, for the case of a null $\mathbf{y.n}$, they do require that any node other than the one pointed to by \mathbf{y} has $r_{y,n}$ evaluated to 0. Therefore, in applying the refining set in Fig. 4 to the structure in Fig. 3(a), our operator *does not* yield the leftmost structure of Fig. 3(b) in the first place, as opposed to the traditional focus operation. (This does not guarantee the integrity of structures resulting from refinement in general, though.)

The refining set of Fig. 4 *does* yield the two rightmost structures in Fig. 3(b). While these are conservative and semantically sane refinements of Fig. 3(a), they are evidently not as tight as could be, as explained in Section 2.4. As this overcompromises the precision of our transformer in this case, we first try to further focus the back \mathbf{p} edge to the node pointed to by \mathbf{y} . A naive solution to this can be found in the form of further sub-case refinement, namely by semantically reducing the refining structures in Fig. 4(b) down to the point where the back \mathbf{p} edge is either 0 or 1, corresponding to the value that $b_{n,p}$ takes for the node

⁵ This condition can be easily enforced using a meet-based precondition filter [2].

pointed to by \mathbf{y} . For example, we can replace the left-hand side structure in Fig. 4(b) by two similar structures, with the difference being that one has both $b_{n,p}(v_1)$ and $p(v_2, v_1)$ evaluating to 1, and the other has their value being 0. This enforces a definite truth value for the back \mathbf{p} edge from the node pointed to by $\mathbf{y.n}$ to the node pointed to by \mathbf{y} in the result of the refinement step.

By applying further reduction in the same style, we can tighten the value of the \mathbf{n} (\mathbf{p}) self-loop on the node that $\mathbf{y.n}$ points to in the middle structure in Fig. 3(b), in correlation with the value of c_n (respectively, c_p) for that node.⁶ However, such a level of enumeration will lead to a number of structures that is exponential in the number of tightened instrumentation predicate values—in this case $b_{n,p}(v_1)$, $c_n(v_2)$, and $c_p(v_2)$ —resulting in 8 disjoint structures. This combinatorial effect gives little hope for scaling a precise enough operator of this kind to cases with even a little more predicate interdependencies. We manage to avoid this explosion in the size of the refining set by exploiting the following properties.

Distributivity of meet over join. As already noted [3], for all sets of structures, $XS \sqcap (XR \sqcup XR') = (XS \sqcap XR) \sqcup (XS \sqcap XR')$. We can therefore split the structures in Fig. 4 such that XR corresponds to Fig. 4(a) and XR' corresponds to Fig. 4(b), with the guarantee that $(\{S\} \sqcap XR) \sqcup (\{S\} \sqcap XR')$ yields the same result as plain meet using the original refining set.

Associativity of meet. As $XS \sqcap (XR'_1 \sqcap XR'_2) = XS \sqcap XR'_1 \sqcap XR'_2$ (the latter being left-associative) for all sets of structures, we can further avoid the combinatorial blow-up in the number of structures needed for the proper reduction, such as the one explained above. Let XR'_1 , XR'_2 , and XR'_3 be the sets containing a pair of structures which reduces the left-hand side structure of Fig. 4(b) with respect to the value of $b_{n,p}(v_1)$, $c_n(v_2)$, and $c_p(v_2)$, respectively. We observe that the elaborate set of reduced refinement structures described above is obtained by $XR'_1 \sqcap XR'_2 \sqcap XR'_3$, as each of these operands requires that $b_{n,p}(v_1)$, $c_n(v_2)$, or $c_p(v_2)$ has a definite value but keeps the others indefinite ($\frac{1}{2}$), respectively. Therefore, $\{S\} \sqcap XR'_1 \sqcap XR'_2 \sqcap XR'_3$ gives us the desired level of tightness, without needing to store the fully expanded set of structures. Note that each of XR'_i , $1 \leq i \leq 3$, consists of exactly two complementary structures. Hence, the successive application of meet operations is likely to reduce the number of unfocused predicates at each step, down to the point where a single fully-tightened structure is obtained.

We can therefore obtain the desired refinement operation by means of

$$(\{S\} \sqcap XR) \sqcup (\{S\} \sqcap XR'_1 \sqcap XR'_2 \sqcap XR'_3) ,$$

for any given structure S . It is important to note that such a formulation in fact shifts the exponential behavior from the size of a single refining structure to the worst case complexity of the additional meet and join operations. Finally,

⁶ For expository reasons, we ignore the case of a two-node cycle, which is also correlated with the value of $c_n(v_2)$ and $c_p(v_2)$.

note that the actual refinement operators used in our framework are fairly more complicated, as they enumerate further instrumentation-implied sub-cases, and are well beyond the scope of this paper.

3.2 Enforcing Integrity

As hinted above, a structure resulting from a structure-based refinement operation does not necessarily satisfy the integrity constraints implied by its instrumentation predicates. In one instance of this problem, a refinement operator yields a structure that has a summary node v for which $r_{y,n}(v) = 1$, but $y(v) = 0$ and $n(v', v) = 0$ for all $v' \neq v$. We consider the use of structure-based filtering to dismiss such a structure prior to the transformation update, mimicking the role of `coerce` in that respect. While arbitrary first-order logic conditions may not necessarily be expressible using 3-valued structures, we can still handle this particular case using our approach. Specifically, it is sufficient here to determine whether a structure has some node that is neither pointed to by \mathbf{y} nor has an inbound \mathbf{n} field pointing from any other node that may be referenced by \mathbf{y} , yet is indicated to be reachable from \mathbf{y} by a sequence of \mathbf{n} references—namely $\exists v. r_{y,n}(v) \wedge \neg y(v) \wedge \forall v'. (y(v') \implies \neg n(v', v))$. Fortunately, a structure S_F consisting of an \mathbf{n} -unreachable summary node, which is not pointed to by \mathbf{y} but denoted with a definite (1) $r_{y,n}$ value, represents this requirement and makes it evaluate to 1.⁷ By the embedding theorem, we have that for any structure that is embedded in S_F , the above formula must evaluate to 1, implying a breached integrity constraint. We therefore apply this test to each structure resulting from a refinement operation, dismissing structures that are embedded in our filter.

4 Loose Embedding

Analyzing the program in Fig. 1 with the framework described so far yields a total of 113 structures, with an average of 3.2 structures per CFG node and a peak of 9 disjoint abstract states for a single node. This large number seems counterintuitive to the actual simple essence of what the program does. In the following, we highlight one source of this inflation and suggest a way to avoid it.

4.1 State-Space Inflation in Loops

Fig. 5 shows three of the abstract structures representing disjoint sets of concrete heap states, arising immediately past the statement $\mathbf{t} = \mathbf{y}.n$ during the analysis of the program in Fig. 1. The structure in Fig. 5(b) represents the set of concrete doubly-linked lists whose head is pointed to by \mathbf{x} , followed by a sequence of (one or more) nodes, followed by a pair of nodes pointed to by \mathbf{y} and \mathbf{t} , respectively, and finally followed by a sequence of (one or more) nodes forming the list suffix.

⁷ In fact, two distinct structures are required to represent all possible configurations corresponding to this case. See [1] for details.

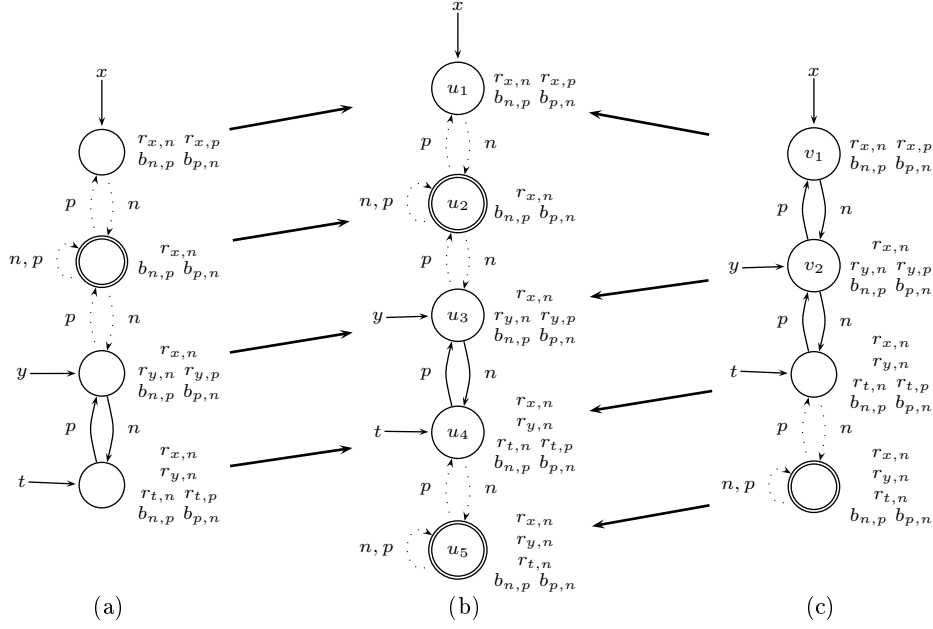


Fig. 5. Abstract heap states arising after the statement $t = y.n$ in Fig. 1.

This structure describes a *general case* that the program exhibits at this program point, providing a conservative approximation of the set of concrete states incurred by the program, and also contributing a general insight regarding the state of computation at this point in the program.

On the contrary, the two other structures in Fig. 5 describe what could be considered a slight variant of the general case. Specifically, Fig. 5(a) represents the set of lists that lack the suffix nodes and Fig. 5(c) represents the set of lists that lack the infix nodes. A fourth structure arising at the same program point—which represents a list with neither infix nor suffix nodes—is not shown.

As is evident from the example, the total number of disjoint abstract states used for representing all possible concrete states is exponential in the number of summary nodes appearing in the general case. The special case descriptors are inevitable by construction of the abstraction framework, given that the loop traverses all nodes of the list. Yet, informally speaking, they seem to contribute very little information compared to what the general case already expresses, consequently fortifying the analysis with only a little precision, but at a high cost.

4.2 Relaxed Definition of Embedding

Recall that the definition of $D_{3\text{-STRUCT}}$ uses a notion of embedding in order to eliminate non-maximal structures, prohibiting expressive redundancy and en-

suming a strict partial order. In attempt to make the special cases of Fig. 5 non-maximal—and therefore disposable—we aim at embedding them into the general case by relaxing the definition of embedding in the following manner.

Allow summaries to represent zero nodes. We allow summary nodes to be excluded from the range of an embedding function, overruling the surjectivity requirement. The individuals of Fig. 5(a) can therefore be mapped to a *subset* of the individuals of Fig. 5(b), as indicated by the bold arrows, excluding only the suffix summary node u_5 from the range of the embedding function. Yet, it is clear that the requirement for predicate interpretation consistency in Definition 3 is satisfied. This allows for the structure in Fig. 5(b) to embed the structure in Fig. 5(a), making the latter disposable.

Retain connectivity via non-mapped summaries. Consider the mapping from individuals of Fig. 5(c) to those of Fig. 5(b), depicted by the bold arrows: the fact that u_2 is excluded from the range of the function breaks the connectivity of the structure in Fig. 5(b) compared to that of the structure in Fig. 5(c). In particular, while $n(v_1, v_2) = 1$ in the former, $n(f(v_1), f(v_2)) = n(u_1, u_3) = 0$ in the latter, prohibiting embedding by this function.

We therefore further permit predicate interpretation consistency of any binary predicate to be checked against the *constrained transitive closure* of that predicate in the target structure, which is only computed via summaries excluded from the range of the function under consideration. Since $n(u_1, u_2) \wedge n(u_2, u_3) = \frac{1}{2}$, the extended consistency requirement is satisfied, making the mapping in the diagram an admissible embedding function.

We now give the formal definition of the relaxed embedding relation.

Definition 5 (Loose embedding). Let $S = (U, \iota)$ and $S' = (U', \iota')$ be two structures and let $f : U \rightarrow U'$ be a function, such that $eq(v, v) = \frac{1}{2}$ for all nodes $v \in V = U' \setminus \text{range}(f)$. We say that f loosely embeds S in S' , denoted $S \sqsubseteq^f S'$, if Eq. (1) holds for all nullary and unary predicates and all nodes, and for every predicate $p \in \mathcal{P}^{(2)}$ and pair of individuals $u_1, u_2 \in U$,⁸

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

S is loosely embedded in S' , denoted $S \sqsubseteq S'$, if there exists f such that $S \sqsubseteq^f S'$.

Note that the above definition immediately extends to the definition of the abstract domain $D_{3\text{-STRUCT}}$ and its associated operators (join and meet), as well as its derived bounded state sub-domain. It also extends to the definition of abstraction and concretization accordingly.⁹

⁸ An empty conjunction evaluates to 1 and an empty disjunction evaluates to 0.

⁹ Note that for the general case of unbounded 3-valued structures, the loose embedding relation induces a partial *preorder*, in turn inducing a preorder on the powerset

4.3 Preserving Soundness

The proposed extensions to the definition of embedding invalidate the foundations of soundness provided by the embedding theorem. We therefore adjust the semantics of logical formula evaluation in accordance with these extensions.

First-order quantification. We interpret each occurrence of the form $\exists u.\phi$ as $\exists u.eq(u, u) \wedge \phi$, assuring that any predicate that is existentially quantified over a summary node is “lowered” to $\frac{1}{2}$. This accounts for the case where no corresponding node exists in some concrete setting, which could cause the formula to evaluate to a (definite) 0. Similarly, we interpret each occurrence of the form $\forall u.P$ as $\forall u.\neg eq(u, u) \vee P$, assuring that any universally quantified predicate is “raised” to $\frac{1}{2}$ for any summary node. This accounts for the absence of corresponding nodes in some concrete setting, which could cause the formula to evaluate to a (definite) 1.

Binary predicate interpretation. We interpret each binary predicate between v_1 and v_2 as the constrained transitive closure of that predicate, namely considering the conjunction of the predicate’s values along any sequence of (zero or more) summary nodes between v_1 and v_2 . As opposed to Definition 5, we cannot consider the set of non-image summaries here, as no embedding function is due. Instead, we consider any summary node for the purpose of transitive closure, but also bound the truth value of such a transitive interpretation by $\frac{1}{2}$ in order to ensure that the result is a conservative approximation with respect to *any* embedding function.¹⁰

The above extensions suffice to retain the soundness of our local transformers, consequently implying global soundness. Note, however, that they also imply potential sources of imprecision as well as added computational effort, especially when transitive binary closure needs to be evaluated. Nonetheless, as binary edges adjacent to summary nodes are commonly indefinite in the first place, we do not expect a significant loss of precision due to the contamination of formula evaluation with $\frac{1}{2}$ values. Also, we expect the excess algorithmic overhead to be absorbed by our highly effective approach for conducting computations over 3-valued structures. Finally, it should be mentioned that loose embedding also deflates some of our structure-based refinement operators, like the two single-node structures in Fig. 4(a), which are now embedded in their respective counterparts.

5 Experimental Results

Table 2 presents analysis statistics for a set of five small Java programs that manipulate singly- or doubly-linked lists, executed on a 1.6GHz Pentium-M, 1GB machine running Linux. This benchmark, along with approximate analysis

domain. We can show it is a strict partial order for the domain of bounded structures, implying that α is still a well-defined function.

¹⁰ For the reasoning behind this additional requirement, see [1].

	stats		reference		specialized strict						specialized loose					
	loc	loop	tot	mem	tot	ave	top	tot	ref	mem	tot	ave	top	tot	ref	mem
sll-loop	33	2	900	1000	109	3.3	9	20	6	88	59	1.8	4	11	5	72
sll-reverse	52	3	3000	2000	226	4.4	9	34	12	188	104	2.0	4	28	14	129
sll-delete	49	3	12400	3200	485	9.9	48	202	59	379	215	4.4	20	100	44	235
dll-loop	35	2	1400	1300	113	3.2	9	27	18	463	61	1.7	4	19	11	441
dll-pairs	42	2	3000	2000	191	4.6	15	69	50	896	105	2.5	8	43	31	846

Table 2. Benchmark results for five Java programs processing singly- and doubly-linked lists. Columns denote program statistics (number of CFG locations and loops), running statistics using TVLA (total analysis time and peak memory consumption), and running statistics using the specialized framework in strict and loose embedding mode (total, average, and maximum number of structures for a CFG node, analysis total and refinement times, and peak memory consumption). Time is in milliseconds, and memory is in kilobytes.

statistics using the TVLA reference implementation on similar hardware, were adopted from [3]. The results suggest several insights regarding the effectiveness of our framework. First, it is shown to converge significantly faster than the reference implementation, ranging from a factor of 40 (using strict embedding on a simple singly-linked list traversal) to a factor of 124 (using loose embedding on a program that deletes an arbitrary element from a singly-linked list). Although an improvement of this kind was expected—our analyzer is restricted by construction and therefore better tweaked for performance—the actual speed-up factor is quite encouraging. While our results do not provide sufficient evidence for the relative effectiveness of structure-based refinement per-se (we implemented neither focus nor coerce in our framework), the fact that the time spent on abstract heap refinement by our analyzer—ranging between 30–73% of the total analysis time—suggests that our structure-based approach is relatively time effective compared to the remaining operations. However, the fact that refinement takes a larger portion in the doubly-linked list case suggests that it may not scale very well as dependencies among predicates increase. Memory consumption is generally lower than that of TVLA, but then again seems not as low in the heavier abstraction (doubly-linked list) as in the lighter abstraction (singly-linked list). Yet this issue has not been the focus of our performance optimization and could probably be improved significantly in the future.

Second, the case of loose embedding appears quite effective in both deflating the number of structures—45–55% and 46–58% deflation in total and top number of structures, respectively—as well as shortening total analysis time, by 17–50%. The case of sll-delete is particularly notable, as one of its loops may terminate abruptly, allowing a greater number of abstract states to “escape” and propagate to other CFG nodes. Here, the use of loose embedding seems to provide the greatest gain in both state set deflation and analysis performance. Finally, it is worth mentioning that the actual (graphical) results of an analysis using loose

embedding are by far more comprehensible—and therefore, more usable—than those of a traditional (strict) analysis. We consider this a nice practical outcome, which supports our view of the problem with strict embedding abstraction.

6 Related Work

This work shares common goals with a few other efforts, all aimed at improving the scalability and applicability of shape analysis to practical uses. In two cases, TVLA powerset heap abstractions were compressed into a single structure [7] or partially disjunctive sets [10] by means of merging (joining) predicate values and allowing individuals to represent zero concrete nodes, or merging structures consisting of isomorphic sets of individuals, respectively. Our loose embedding approach seems to resemble the former to some extent, as both allow certain nodes to represent zero concrete nodes and use a relaxed notion of first-order quantification in formulas. It also shares a similar approach to the latter, as both attempt to reduce the number of structures describing “similar” cases based on some criteria. Nonetheless, by carefully defining the notion of descriptive redundancy, and by extending the definition of the embedding relation—rather than overloading predicates or joining structures—our approach has the advantage of not inducing imprecision on remaining representative abstract states.¹¹

Other approaches deviate from 3-valued canonical abstraction and examine the use of predicate abstraction for analyzing shape properties [11,5]. While such approaches were shown to yield precise and descriptive results, their contribution to scalability of shape problems is unclear due to the larger number of predicates required for sufficient precision.

It is well-known that structure-based semantic reduction is expressively inferior to general FOL formula-based refinement (*i.e.*, as obtained by focus and coerce [12]). However, we argue that precision can be improved by further elaboration of refining structures, suggesting a trade-off between sufficient precision and tolerable complexity. As far as we know, our work is the first attempt to deploy practical structure-based abstraction refinement for 3-valued logic shape analysis and may serve as a point of reference for future efforts.

7 Conclusion

We described a new implementation of a 3-valued logic-based shape analysis tool that uses an effective structure-based approach for refining abstract heaps, and deflates abstract state sets using an alternate definition of structure ordering. We applied it to a small set of benchmark programs, with encouraging results, regarding both the effectiveness of the analysis framework as well as the successful restraining of powerset abstract states exhibited by the analysis. We believe

¹¹ Note that empty summaries have been used by other analyses (*e.g.*, [6]), so the novelty of our approach in this context is restricted to 3-valued logic-based analysis.

that the next step in this direction is to further extend and examine the applicability of our analyzer to different (and more complex) heap structures on the one hand, and to assess the usefulness of loose embedding for programs of higher complexity in attempt to assert its expected advantages on the other. A separate effort, aimed at the automatic derivation of sound and precise structure-based refining operators from logical formulas, should guarantee the correctness and improve the adaptivity of our approach, and is currently a work in progress.

Acknowledgments

I thank Mooly Sagiv and Roman Manevich for their useful feedback and continuous advice. I also thank Ras Bodik for supporting my work on this research.

References

1. G. Arnold. Lightweight specialized 3-valued logic shape analyzer. Technical Report UCB/EECS-2006-59, EECS Department, University of California, Berkeley, May 2006. Available at <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-59.html>.
2. G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Intersecting heap abstractions with applications to compile-time memory management. Technical Report TR-2005-04-135520, Tel-Aviv University, Apr. 2005. Available at <http://www.cs.tau.ac.il/~rumster/TR-2005-04-135520.pdf>.
3. 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.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symposium on Principals of Programming Languages (POPL)*, pages 269–282. ACM Press, 1979.
5. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Symposium on Principals of Programming Languages (POPL)*, pages 115–126, 2006.
6. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *The European Symposium on Programming (ESOP)*, volume 3444, pages 124–140. Springer-Verlag, 2005.
7. T. Lev-Ami. TVLA: A framework for kleene logic based static analysis. Master's thesis, Tel-Aviv University, May 2000.
8. T. Lev-Ami, T. W. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 26–38, 2000.
9. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS)*, pages 280–301, 2000.
10. R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Static Analysis Symposium (SAS)*, pages 265–279, 2004.
11. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 181–198, 2005.
12. M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.