Lightweight Specialized 3-Valued Logic Shape Analyzer

Gilad Arnold

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-59
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-59.html

May 16, 2006

Acknowledgement

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

Lightweight Specialized 3-Valued Logic Shape Analyzer

Gilad Arnold

May 16, 2006

Abstract

We describe the design and implementation of a specialized shape analysis tool based on 3-valued logic. Our analyzer can reason about programs manipulating recursive data structures, such as singly- and doubly-linked lists, yielding precise results that are comparable to those of the well-known reference implementation, in only a fraction of the time. In particular, (a) we apply a new and effective technique for performing structure-based abstraction refinement, and (b) introduce a new definition for ordering of abstract heap descriptors which leads to a significant deflation in the sets of abstract states explored by the analysis. Although currently restricted by design in its applicability for different flavors of analyses, we argue that—thanks to a modular and extendable architecture—generalizing the capabilities of our tool can be achieved by means of further mostly straightforward (yet non-trivial) software engineering effort.

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 [6]; and checking properties of heap references throughout the program, such as dead objects analysis and its applications to static garbage collection [2]. Shape analysis also generalizes alias analysis, and therefore can replace less precise techniques such as pointer analysis, that are traditionally used for this purpose.

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 use of conservative abstraction methods following [3] yields 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.

```
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;
}
```

Figure 1: A Java program which constructs and traverses a doubly-linked list

1.1 3-Valued Logic Shape Analysis

We consider a shape analysis framework which models heap topology and related properties using logical structures, and applies first-order logic formulas to model program semantics [11]. 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 TVLA [7] 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 obviously underoptimized compared to a (presumed) specialized implementation. Still, we can observe at least two aspects which make a reference implementation inherently cumbersome.

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

State set inflation. The huge abstract domain underlying the analysis—whose induced complexity is double-exponential by 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.

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 [7] yields a total of 113 abstract heap structures and takes 1.4 seconds to complete with stock TVLA. A slightly more complicated example—a singly-linked list manipulating program that has 3 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.

1.2 Main Results

This paper describes the fresh implementation of a 3-valued logic based shape analysis tool. Our analyzer is intently restricted compared to the fully-parameterized reference implementation, but appears to be better suited for performance and scalability thanks to a graph-based representation of structures and few other enhancements, which naturally lend themselves to implementation of efficient analysis-specific algorithms such as computing meet/join operations, computing transitive closure of binary predicates, and applying semantic updates to 3-valued structures. In particular, our framework suggests the following major contributions and artifacts.

Specialized structure-based refinement. While using a meet operator for abstraction refinement has already been suggested [2] 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. Technically, we were 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 for the represented set of concrete states. Consequently, we propose an alternate definition of embedding of 3-valued logical structures by which "special case" descriptors are unnecessary (redundant) in the presence of a "general case", by allowing abstract summary heap nodes to represent no concrete nodes at all, yet still retain connectivity between other elements of the heap in a conservative manner. This extension 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, showing an overall speed-up of up to 124 and an up to 15 times smaller memory footprint.

1.3 Outline

The rest of this paper is organized as follows. In Section 2 we introduce 3-valued logic based heap abstraction, and bound the scope of the analyses addressed

Predicates	**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
b_{f_1,f_2}	The f_2 field of an object pointed by the f_1 field of v points back to v

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

by this work. Section 3 outlines the design and implementation of our shape analysis framework, with structure-based refinement being explained in greater detail in Section 4 . Section 5 describes the loose embedding approach for constraining the effect of abstract state set inflation. Initial experimental results are given in Section 6. Section 7 discusses related work, and Section 8 concludes.

2 3-Valued Logic Shape Analysis

We explain the heap state abstraction and abstract transformers following [11], which underly our technique for static shape analysis.

2.1 Concrete Program States

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

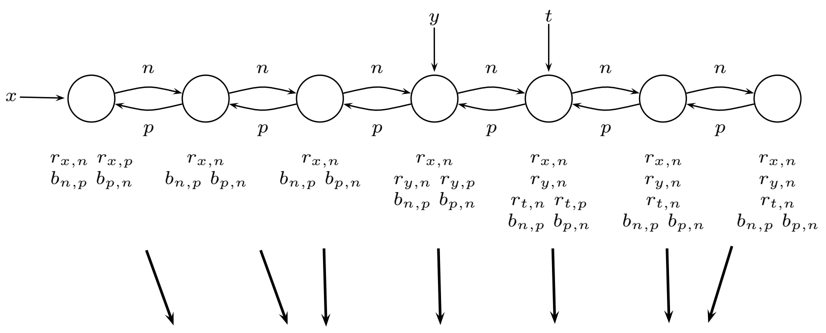
Definition 1 (Concrete state). A 2-valued logical structure over a vocabulary (set of predicates) $\mathcal{P}$ is a pair $S = (U, \iota)$ where U is the universe of the 2-valued structure, and ι is the interpretation function mapping predicates to their truth-value in the structure: for every predicate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

We require that the set of predicates includes the binary predicate *eq*, bearing the semantics of *equality* between individuals. Table 1 shows the set of predicates used in the analysis of the program in Fig. 1, with p and f instantiated over $\{\mathrm{x, y, t}\}$ and $\{\mathrm{n, p}\}$, respectively.[1] 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 [7, 11].

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

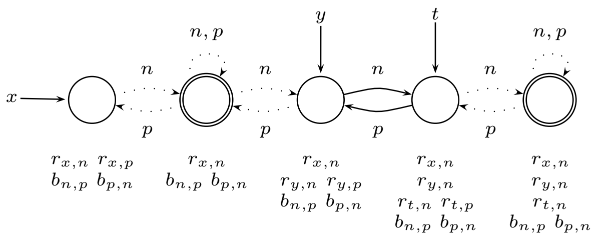
[1]Note that b_{f_1,f_2} is only instantiated for pairs of *distinct* reference fields.

(a)



x
y
t
n
p
$r_{x,n}$ $r_{x,p}$ $b_{n,p}$ $b_{p,n}$
$r_{x,n}$ $b_{n,p}$ $b_{p,n}$
$r_{x,n}$ $r_{y,n}$ $r_{y,p}$ $b_{n,p}$ $b_{p,n}$
$r_{x,n}$ $r_{y,n}$ $r_{t,n}$ $r_{t,p}$ $b_{n,p}$ $b_{p,n}$
$r_{x,n}$ $r_{y,n}$ $r_{t,n}$ $b_{n,p}$ $b_{p,n}$

(b)



x
y
t
n
p
n, p
$r_{x,n}$ $r_{x,p}$ $b_{n,p}$ $b_{p,n}$
$r_{x,n}$ $b_{n,p}$ $b_{p,n}$
$r_{x,n}$ $r_{y,n}$ $r_{y,p}$ $b_{n,p}$ $b_{p,n}$
$r_{x,n}$ $r_{y,n}$ $r_{t,n}$ $r_{t,p}$ $b_{n,p}$ $b_{p,n}$
$r_{x,n}$ $r_{y,n}$ $r_{t,n}$ $b_{n,p}$ $b_{p,n}$

Figure 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)

to some node v is depicted as 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 drawn as a directed edge from v_1 to v_2, and labeled f. 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 `t = 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-$STRUCT[\mathcal{P}]$, 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 [11], 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.

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

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 double-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-$STRUCT \subsetneq$ 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 $f : U \to U'$ 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 $u_1, \ldots, u_k \in U$,

$$p^S(u_1, \ldots, u_k) \sqsubseteq p^{S'}(f(u_1), \ldots, f(u_k)) \ . \tag{3.1}$$

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

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

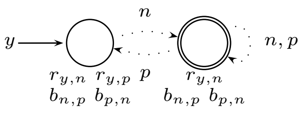
In the following, we restrict sets of 3-valued structures by disallowing non-maximal structures. This ensures that the above Hoare order is indeed a proper partial order. The set $D_{3\text{-}STRUCT}$, consisting of all finite sets of 3-valued structures that do not contain non-maximal structures, along 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 fundamental abstraction method [11] 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 $D_{B\text{-}STRUCT}$ 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 *canonical abstraction* [11]—maps a potentially unbounded 2-valued structure into a bounded 3-valued structure. Namely, $\beta\big((U, \iota)\big) = (U', \iota')$, where U' consists of the disjoint subsets of U for

(a)



(b)

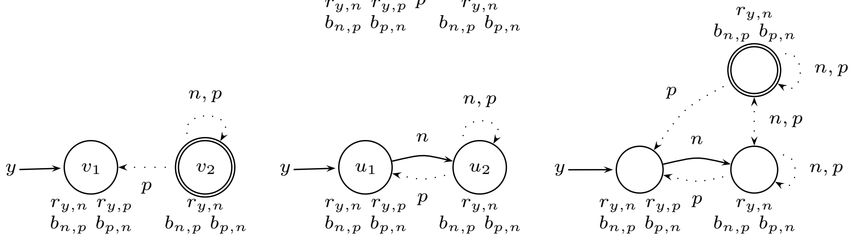


Figure 3: Structure refinement using focus: (a) the original inbound structure; (b) structures resulting from focus using the formula $\exists v'.y(v') \wedge n(v', v)$

which no unary predicate evaluates to distinct definite values, and the interpretation ι' of each $p \in \mathcal{P}^{(k)}$ and k individuals $c_1, \ldots, c_k \in U'$ is given by

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

Fig. 2(b) shows 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 [11] 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 `t = y.n` from the example in Fig. 1 is modeled by $t(v) \leftarrow \exists v' : y(v') \wedge n(v', v)$. The *embedding theorem* [11] ensures that the result of a transformation on any abstract state is a sound approximation of the best transformer.[3] 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,[4] we achieve *partial concretization* by means of two auxiliary operations [11].

Focus. Given a first-order logic formula ϕ, this operation performs semantic reduction of a 3-valued structure such that the evaluation of ϕ on any resulting structure 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 `t = y.n`—using the formula $\exists v'.y(v') \wedge$

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

[3]Note that concretization due to the best transformer is such that guarantees further *integrity constraints* [11], implied by the interpretation of instrumentation predicates.

[4]Recent work [12] has deployed a theorem prover in order to elevate the applicability of best transformers for the case of 3-valued logic abstraction.

$n(v', v)$. 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), where v_2 represents concrete nodes reachable from `y` by a sequence of `n` fields, yet all outbound `n` references from v_1 are interpreted to be 0. Also, focus might yield structures that are not as precise as could be with respect to the values of instrumentation predicates, such as the middle structure in Fig. 3(b), whose back `p` edge from u_2 to u_1 could be tightened to 1, and whose self-loop `n` and `p` edges on u_2 could be tightened to 0.

Coerce. The functionality of this operator 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.

A refinement process using the above steps is guaranteed to retain soundness of abstract states—up to the consistency of the semantics of integrity rules—for any given focus formula and abstraction instrumentation. It should be noted that the above—in particular the coerce step—are by far the most time consuming phases of the analysis in practice, suggesting that an alternate approach may be highly improve efficiency.

3 Specialized 3-Valued Logic Analyzer

We describe the characteristics of our implementation of a restricted variant of the parametric 3-valued domain analysis framework in [11].

3.1 Restricted Abstraction

Aiming at a fast and practical analysis tool, we restrict ourselves to a "skim" and specialized variant of the 3-valued logic abstraction, which we believe can provide a good trade-off between versatility and performance.

Limited predicate arity. As opposed to TVLA, in which predicates in the abstraction can be defined to be of any arity $k \geq 0$, we restrict our framework to only support nullary, unary, and binary predicates. This is guided by the fact that the heap relations modeled by shape analysis always correspond to either of those types of predicates: method-local reference variables are normally modeled by unary predicates,[5] and reference fields are modeled by binary predicates. We also want to model Boolean variables and fields, in order to increase the precision of our abstract interpreter:

[5] In this context, it is worth mentioning that one approach for interprocedural shape analysis extends this notion, by modeling local references as binary relations between stack objects (frames) and heap objects.

local Boolean variables are modeled by nullary predicates, and Boolean object fields are modeled by unary predicates. In the case of our specialized abstraction, all instrumentation properties are modeled by unary predicates (see below).

Fixed abstraction instrumentation. Our proposed implementation supports a fixed set of instrumentation predicates with predefined semantics, that can be either instantiated or not during the initialization of the analysis. The supported predicates includes the ones shown in Table 1.

Reachability. For each pair consisting of a (unary) predicate x representing a reference variable, and a (binary) predicate f representing a reference field, a (unary) instrumentation predicate $r_{x,f}$ can be generated. If $r_{x,f}(v)$ holds, then v represents an object that is reachable from reference variable `x` through a sequence of (zero or more) `f` dereferences. Note that tracking the value of such a predicate might require an explicit evaluation of transitive binary closure, to be discussed later.

Cyclicity. For a (binary) predicate f representing a reference field, a (unary) instrumentation predicate c_f can be generated. If $c_f(v)$ holds, then v represents an object which resides on a cycle formed by a sequence of (one or more) `f` references.

Sharing. For a (binary) predicate f representing a reference field, a (unary) instrumentation predicate s_f can be generated. If $s_f(v)$ holds then v represents an object that is being pointed from more than one object through an `f` field.

Back-reference. For a pair of (binary) predicates f_1, f_2 representing reference fields, a (unary) instrumentation predicate b_{f_1,f_2} can be generated. If $b_{f_1,f_2}(v)$ holds then v represents an object whose reference to some object via an `f1` field implies a back-reference via an `f2`.

The above set has been used as a standard abstraction for precise reasoning in various shape related tasks. While more complicated instrumentation properties may lead to greater precision—binary reachability instrumentation is one example—we chose not to support them currently. Also note that application specific predicates, used for proving more delicate properties of heap structures—e.g., field or object liveness, sortedness of objects, and so on—can be added on top of the existing abstraction at a later phase. With loss of generality, we currently only care for the general purpose abstraction.

Although in this framework we do not implement automated inference of instrumentation values—which can be derived from their declarative definition by means of *finite differencing* [10]—the fact that we implement a fixed set of transformers implies that both correctness and precision of

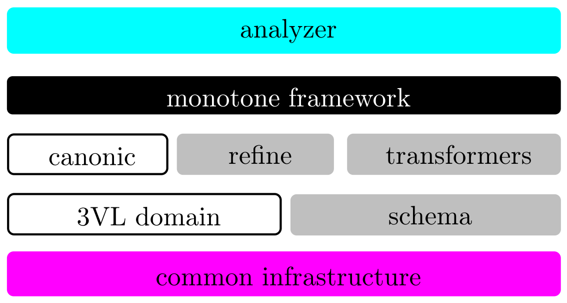


Figure 4: Block diagram of the specialized analyzer implementation

their value assignment is controlled by the framework itself, and implemented within the fixed set of provided transformers (see below).

Hard-coded transformers. Similar to the above, our proposed framework supports a fixed set of transformers which we believe to suffice for modeling the semantics of common imperative languages with pointers and destructive updates. Specifically, we support an empty statement; assignment of either a null value, a reference variable, or a field dereference, to either a reference variable or field; conditional control statement with reference equivalence or Boolean values; object allocation and deallocation; and procedure call/return statements. We assume the use of a simplifying front-end to yield a canonical intermediate form conforming to this set of transformations. At this point we do not support array operations, and our current support for procedure call/return does not implement context-sensitivity, which we consider future work.

Our set of transformers is derived from a mature TVLA analysis specification, which is believed to be sound and quite precise with respect to the abstraction instrumentation. The details of the abstract transformer semantics are omitted for brevity.

Note that the above assumptions do not pose a constant restriction on the generality of our implementation, other than the one bounding the arity of predicates, as all others are transient constraints which can be relaxed at a later time.

3.2 Architecture

The block diagram in Fig. 4 shows the architecture of our framework. Blocks having the same color indicate a related or dependent functionality. As shown, our implementation of the 3-valued abstraction domain, along with that of canonical abstraction, are independent from any particular analysis layout. The latter is defined using the analysis-dependent modules, including the schema (responsible for predicate definition and layout, construction of analysis-specific instrumentation scheme, etc.), the refinement module (responsible for constructing and applying refinement operators), and the associated abstract transformers module. A monotone framework front-end provides a unified interface to a singleton analysis instance. A simple analyzer, capable of parsing TVLA-like specification, constructing an analysis, and conducting the chaotic iterations algorithm, is provided for evaluation purposes.

The above diagram suggests that further extensions to the analysis engine, like support for additional abstraction or instrumentation features, can be undertaken without needing to modify the abstract domain implementation. Similarly, optimization of domain related operators and algorithms can be carried out independently from overlying abstraction semantics.

3.3 3-Valued Logic Abstraction Domain

We describe the outline of our topology-based approach for representing and manipulating 3-valued structures and powerset elements.

3.3.1 3-valued structures

We represent 3-valued structures using directed graphs with attribute vectors attached to graph nodes and edges, corresponding to values of unary and binary predicate, respectively. Nullary predicate values are stored separately.

The use of a graph-based representation has several justifications. Initially, it is a concise representation exploiting the sparsity of concrete (and abstract) heap configurations: as evident from abstract transformer semantics, the computation of most updates only cares for non-false predicate values, hence a sparse data structure is likely to be an advantage. Furthermore, we believe that the greatest advantage lies within the fact that—unlike structure-oblivious hash-based representation—the preservation of actual structure topology promotes faster evaluation of transformer updates due to the ability to easily extract sequences of “heap paths”. For example, consider the following formula used for updating the value of a reference variable predicate when applying the transformer corresponding to `t = y.n`,

$$t(v) \leftarrow \exists v' : y(v') \wedge n(v', v) \ .$$

A naive evaluation (associated with a structure-oblivious representation) would have to consider every possible pair of nodes in order to evaluate the conjunction. On the contrary, a sparse topology-aware representation may consider, for each node v' that is referred to by y, only the subset of nodes $\{v \mid v \in U\}$ for which $n(v', v)$ is not false (i.e., there exists a graph edge with non-false n attribute). Since structures are likely to be sparse, the resulting evaluation process is expected to be more economical than an exhaustive one. Put in a general way, it is expected to promote faster computation wherever an existential formula is used to infer non-false binary connectivity, a common case in the analysis transformations specification.

Our representation is also likely to minimize overhead when transitive closure of binary relations needs to be computed. For example, consider the following formula used for updating the value of reachability instrumentation predicates upon an assignment statement `t = y.n`,

$$r_{t,f}(v) \leftarrow \begin{cases} r_{y,f} \lor (\exists v' : y(v') \land n(v', v)) \\ \quad \text{if } \exists v', v'' : y(v') \land b_{n,f}(v') \land n(v', v''), \\ \exists v', v'' : y(v') \land n(v', v'') \land f^*(v'', v) \\ \quad \text{otherwise,} \end{cases}$$

evaluated for each reference field $f \neq n$ for which $r_{y,f}$ is defined. (Note, that due to the Kleene way of logical value interpretation, it might be the case that both sub-formulas need to be evaluated.) Here, the notation f^* stands for a transitive conjunctive closure of the binary predicate f over sequences of pairs of individuals: while the naive way of evaluation (associated with a structure-oblivious representation) would have to consider all the nodes on each iteration of the transitive expansion, our implementation can compute a complete closure in strictly linear time.[6]

3.3.2 Domain operators

Our framework implements the join and meet operators, as well as the embedding relation ($\sqsubseteq$), based on a fast heuristic-based method for computing structure embedding and correspondence relations, as described in [1]. The general approach applies a 3-phase algorithm, in which (a) a matching graph approximating the set of all possible relations (i.e., correspondence relations or embedding functions) is constructed based on unary *predicate consistency*; (b) exhaustive enumeration of all possible generalized perfect matchings is performed using an effective heuristic strategy; then (c) full consistency of higher-arity predicates is asserted for each matching, asserting the required relation (embedding) or generating a set of structures (meet).

Making the matching procedure parametric by the matching quotas associated with nodes, we are able to find candidate relations for either meet structures—requiring that any node is matched exactly once (non-summaries) or at least once (summaries) at each operand structure—or embedding relations, requiring that an embedded structure's nodes are matched exactly once. We require a logical consistency to be either that of agreement (meet) or embedding (join), accordingly. Note that this flexibility also allows us to easily re-use our matching procedure to decide and infer different relations, such as the ones implied by the loose embedding adjustments of Section 5: all it takes is changing the matching quotas such as to allow summary nodes to be matched *zero* or more times, and extend the consistency checking procedure for the case of loose embedding accordingly.

[6]Moreover, in the case of the above formula, our specialized transformer implementation combines both efficient existential quantification and a fast transitive closure computation, yielding a rather economical evaluation process compared to the exhaustive one.

While the matching procedure underlying the enumeration of either relation is quite efficient, our experience has been that the operators' implementation spends most of its time in constructing matching graphs, validating matching results, and constructing intermediate structures (in the case of meet). The former hints that an efficient representation of node attributes (unary predicates), such that supports fast Kleene operators over whole attribute vectors, is highly desirable. The other two strengthen this intuition, but also hint that a topology-aware representation of binary predicate interpretation may promote the validation and generation of intermediate structure. We therefore expect our implementation to improve in that respect as well.

Our implementation of the bounded 3-valued structures powerset domain strictly complies with the definitions in Section 2, ensuring that the implementation of meet and join following [1] yields precise values with regard to the domain of bounded structures that we use. Furthermore, although both meet and join were shown to be hard problems over the domain of unbounded 3-valued structures, we are able to infer them both in a surprisingly effective manner. In particular, our algorithm decides embedding of canonically bounded structures in strictly polynomial time.[7]

3.4 Analysis Specification

Our framework deploys an effective technique for representing an analysis' semantic specification, and applies a new method for achieving fast and precise abstraction refinement without the use of dedicated, general-purpose semantic refinement algorithms mentioned in Section 2.4. The latter technique is explained in detail in Section 4.

We provide an infrastructure for constructing per-program analysis, by means of adding predicates that correspond to program-specific variables and fields. While this is pretty straightforward, we use a graph-based representation for capturing the relations between those predicates and their induced instrumentation. For example, assuming a schema which contains a single unary predicate x and two binary predicates f_1 and f_2, we represent their interrelated instrumentation predicates so that they can be effectively retrieved and traversed when applying transformation updates. This way, an instrumentation predicate r_{x,f_1} is attached to the directed edge (x, f_1), instrumentation predicates b_{f_1,f_2} and b_{f_2,f_1} are attached to directed edges (f_1, f_2) and (f_2, f_1), respectively, and instrumentation predicates c_{f_1} and s_{f_2} are attached to nodes f_1 and f_2, respectively. This layout promotes faster access to predicates that are relevant to an update transformation, as in the case of the update formula for $r_{x,f}$ in Section 3.3.1, applied to each $f \neq n$ for which $r_{x,f}$ is defined.

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

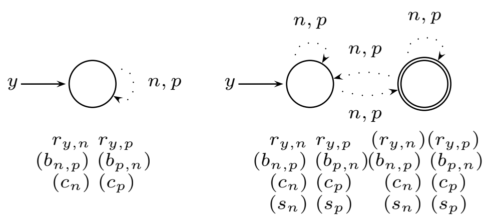

y
n, p
$r_{y,n}$ $r_{y,p}$
$(b_{n,p})$ $(b_{p,n})$
(c_n) (c_p)
y
n, p
n, p
n, p
n, p
$r_{y,n}$ $r_{y,p}$ $(r_{y,n})(r_{y,p})$
$(b_{n,p})$ $(b_{p,n})(b_{n,p})$ $(b_{p,n})$
(c_n) (c_p) (c_n) (c_p)
(s_n) (s_p) (s_n) (s_p)

Figure 5: A (simplified) structure-based filter requiring that `y != null`

4 Structure-Based Refinement

The fact that a meet operator can be used to perform abstraction refinement—both focusing an abstract structure prior to update, as well as filtering structures based on some semantic condition—has already been discussed elsewhere [1]. 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 exemplify this approach by describing a simplified structure-based refinement operator for the abstract transformer of `t = y.n` from Fig. 1. This refinement operator—requiring that the `n` field of the object pointed by `y`, and the `y` reference itself, are definite—is by far the most complicated one, mainly due to the subtle subcases that need to be considered for obtaining sufficient precision.

4.1 Sufficiently Tight and Effective Refinement

We first require that any semantically sane input structure is such whose y predicate evaluates to 1 for exactly one individual (i.e., the dereference `y.n` must succeed). This condition can be easily enforced via a meet-based precondition using the set of structures shown in Fig. 5 (with the simplification of ignoring x and t, and their induced instrumentation).

For the actual purpose of refining the `y.n` dereference, we can initially use a set of structures consisting of the three distinct cases where `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 yet to be split into disjoint subcases, indicating whether additional nodes—other than the one pointed by `y` and (possibly by) `y.n`—may exist. Such a refinement set (again with the simplification of ignoring x and t), is shown in Fig. 6.

In this example, the refining structures impose very few constraints on the values of binary predicates between the different nodes—other than focusing the `n` field of the object pointed by `y`—and, consequently, on those of instrumentation predicates. Still, for the case of a null `y.n` (the two uppermost structures of Fig. 6(a)) they do require that any node other than the one pointed by `y` has $r_{y,n}$ evaluated to 0. Therefore, in applying the refining set in Fig. 6 to

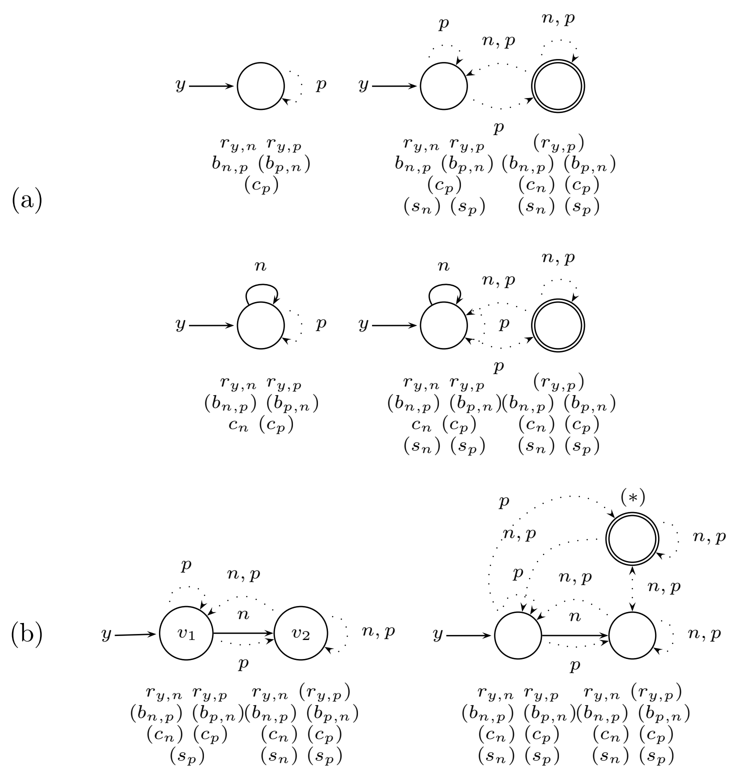


Figure 6: A (simplified) structure-based refining set for `t = y.n`: (a) cases of null or self-loop `y.n`; (b) cases of `y.n` pointing to a different node

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 ensure the integrity of structures resulting from refinement in general, though.)

The refining set of Fig. 6 *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 over-compromises the precision of our transformer in this case, we first try to further focus the back `p` edge to the node pointed by `y`.

A strawman solution to this can be found in the form of further sub-case refinement, namely by semantically reducing the refining structures in Fig. 6(b) down to the point where the back `p` edge is either 0 or 1, corresponding to the value that $b_{n,p}$ takes for the node pointed by `y`. For example, we can replace the left-hand side structure in Fig. 6(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 `p` edge from the node pointed by `y.n` to the node pointed by `y` in the refinement result.

By applying further reduction in the same style we can tighten the value of the `n` (`p`) self-loop on the `y.n` pointed node of the middle structure in Fig. 3(b), in correlation with the value of c_n (respectively, c_p) for that node.[8] However, such a level of enumeration will lead to a number of structures that is exponential by 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 slightly more predicate interdependences. 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 in [2], 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. 6 such that XR corresponds to Fig. 6(a) and XR' corresponds to Fig. 6(b), with the guarantee that $(\{S\} \sqcap XR) \sqcup (\{S\} \sqcap XR')$ yields the same result as plain meet using the whole 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 avoid the combinatorial blow-up in the number of structures resulting from the further reduction of the structures of Fig. 6(b), as explained above. Here we exemplify this for the leftmost structure of Fig. 6(b) only: let XR'_1, XR'_2, and XR'_3 be the sets containing a pair of structures as shown in Fig. 7(a), Fig. 7(b), and Fig. 7(c), 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 ($\frac{1}{2}$). Therefore, $\{S\} \sqcap XR'_1 \sqcap XR'_2 \sqcap XR'_3$ gives us the desired level of tightness,

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

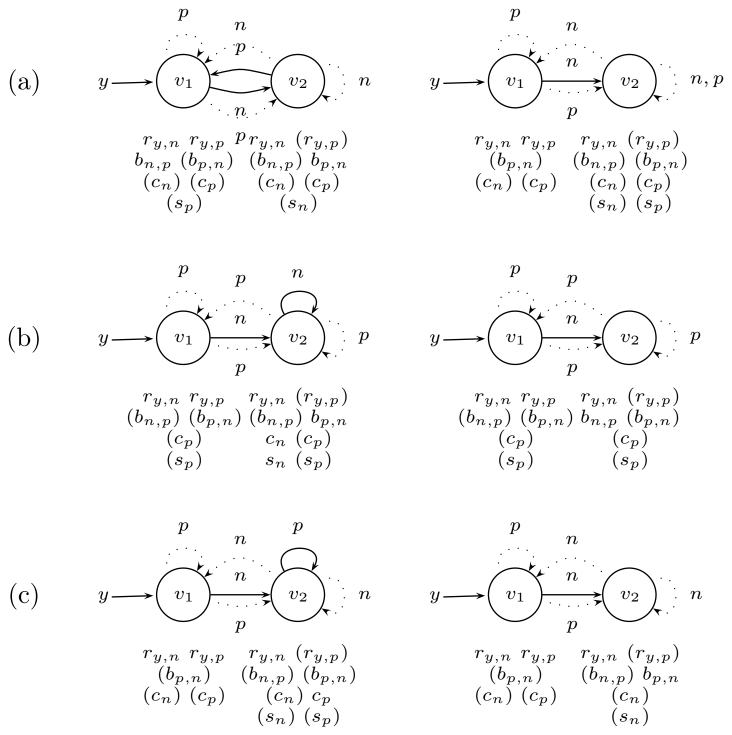


Figure 7: A tuple of (simplified) reductions of the leftmost structure in Fig. 6(b) with respect to the value of (a) $b_{n,p}(v_1)$, (b) $c_n(v_2)$, and (c) $c_p(v_2)$

without needing to store the fully expanded set of structures. Note that any of XR'_i 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, keeping our refining sets of structures at a total linear size.

Few observations apply to this approach: first, 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, yet we expect it to result in an effective computation in practice thanks to the "pruning" nature of a staged refinement of this kind, and the practical effectiveness of our operators. Second, it is important to note that any of the reductions shown in Fig. 7 affects the values of predicates other than the particular predicate being tightened, suggesting an even further tightening on each step in the refinement process. Third, while a manual process of partial reductions suggests possible sources of unsoundness, we can generally obtain the refining sets by applying safe refinement operations to an initial refining structure, this way ensuring correctness without involving runtime invocation of such algorithms. Such an approach is not implemented currently.

Finally, we should comment that the actual refinement operators used in our framework are significantly more complicated, as they enumerate further implied sub-cases, but otherwise are mostly technical and tedious.

4.2 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 such a problem, a refinement operator yields a structure that has a summary node whose $r_{y,n}$ interpretation evaluates to 1, but all inbound n interpretations evaluate to 0. We consider the use of structure-based filtering for dismissing such a structure prior to the transformation update, mimicing 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, here it is sufficient to determine whether a structure has some node that is neither pointed by `y` nor has an inbound `n` field pointing from any other node which may be referenced by `y`, yet for which $r_{y,n}$ evaluates to 1—namely $\exists v. r_{y,n}(v) \wedge \neg y(v) \wedge \forall v'.(y(v') \implies \neg n(v', v))$. The set of structures shown in Fig. 8 represents this condition (with the simplification of ignoring x and t, and their induced instrumentation). By the embedding theorem we have that, for any structure embedded in any of these structures, the above formula

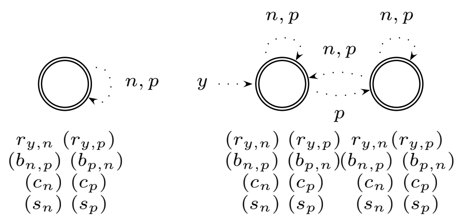


Figure 8: A (simplified) structure-based filter for capturing structures containing a node for which $r_{y,n}$ evaluates to 1, but otherwise is referenced by neither `y`, nor an `n` field from a node referenced by `y`

must evaluate to 1, implying a breached integrity constraint. We therefore check embedding of each structure resulting from a refinement operation against the above set, dismissing structures for which such a relation exists.

5 Loose Embedding

Analyzing the program in Fig. 1 with our 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.

5.1 State-Space Inflation in Loops

Fig. 9 shows three of the abstract structures representing disjoint sets of concrete heap states, arising immediately past the statement `t = y.n` during the analysis of the program in Fig. 1. The structure in Fig. 9(b) represents the set of concrete doubly-linked lists whose head is pointed by `x`, followed by a sequence of (one or more) nodes, followed by a pair of nodes pointed by `y` and `t`, respectively, and finally followed by a sequence of (one or more) nodes forming the list suffix. 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 the computation at this point in the program.

On the contrary, the two other structures in Fig. 9 describe what could be considered a slight variant of the general case. Specifically, Fig. 9(a) represents the set of lists that lack the suffix nodes and Fig. 9(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 by 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

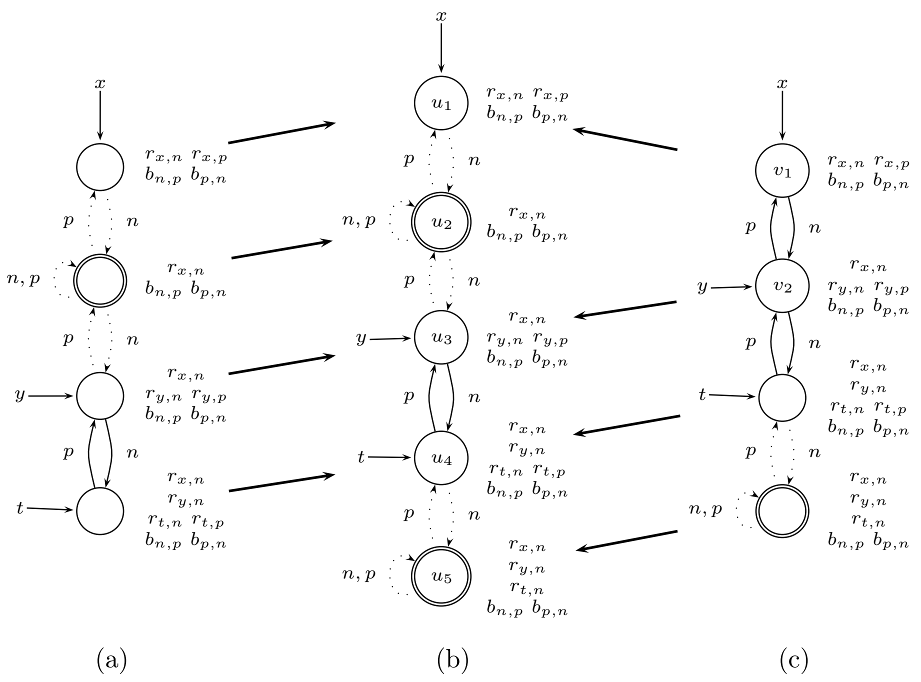


Figure 9: Abstract heap states arising after the statement `t = y.n` in Fig. 1.

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

5.2 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, this way prohibiting expressive redundancy and ensuring a strict partial order. In attempt to make the special cases of Fig. 9 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. 9(a) can therefore be mapped to a *subset* of the individuals of Fig. 9(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. 9(b) to embed the structure in Fig. 9(a), making the latter disposable.

Retain connectivity via non-mapped summaries. Consider the mapping from individuals of Fig. 9(c) to those of Fig. 9(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. 9(b) compared to that of the structure in Fig. 9(c). In particular, while $n(v_1, v_2) = 1$ in the former, $n(f(v_1), f(v_2)) = n(u_1, u_2) = 0$ 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, such that 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 above relaxed embedding relation.

Definition 5 (Loose embedding). Let $S = (U, \iota)$ and $S' = (U', \iota')$ be two structures and let $f : U \to U'$ 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 $S \mathrel{\widetilde{\sqsubseteq}^f} S'$, if (3.1) holds for all nullary and unary predicates and all nodes, and for every predicate $p \in \mathcal{P}^{(2)}$ and a pair of individuals $u_1, u_2 \in U$,[9]

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

We say that S is *loosely embedded* in S', denoted $S \mathrel{\tilde{\sqsubseteq}} S'$, if there exists a function f such that $S \mathrel{\tilde{\sqsubseteq}^f} S'$.

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 sub-domain of bounded states. It also extends to the definition of abstraction and concretization, accordingly.

5.3 Abstraction Monotonicity

We first observe that loose embedding indeed *relaxes* traditional embedding, and that it induces a transitive relation. The former follows from the fact that Definition 5 generalizes Definition 3, therefore a traditionally embedded pair is also loosely embedded using the same function. As for the latter, by composing two functions loosely embedding S_1 in S_2 and S_2 in S_3, respectively, we get a third function which loosely embeds S_1 in S_3: the consistency of predicate interpretations follows from the transitivity of Kleene logic value ordering.

Note that loose embedding is not antisymmetric for the case of general 3-valued structures, therefore induces a partial *preorder*, in turn implying a powerset preorder relation. However, we can show it is a strict partial order for the domain of bounded structures, implying that α is still a well-defined function.

Lemma 6. *Let $S_1 = (U_1, \iota_1), S_2 = (U_2, \iota_2) \in D_{B\text{-}STRUCT}$ such that $S_1 \mathrel{\tilde{\sqsubseteq}^{f_1}} S_2$ and $S_2 \mathrel{\tilde{\sqsubseteq}^{f_2}} S_1$. Then S_1 and S_2 are structurally isomorphic.*

Proof. We first argue that S_1 has an empty universe iff S_2 has an empty universe, implying an isomorphism. Otherwise, let $v \in U_1$ be some individual such that $f_1(v) = u \in U_2$ and $f_2(u) = v'$. By transitivity of Kleene logical values it follows that $p(v) \sqsubseteq p(v')$ for any unary predicate $p \in \mathcal{P}^{(1)}$. Therefore, by the definition of bounded states we have that $v = v'$, hence $f_2 = {f_1}^{-1}$, and both functions are surjective. It follows that $S_1 \sqsubseteq^{f_1} S_2$ and $S_2 \sqsubseteq^{f_2} S_1$, hence by antisymmetry of traditional embedding we have that S_1 and S_2 are isomorphic. $\square$

We now state the monotonicity of our abstraction with the extension for loose embedding.

[9]We assume that an empty conjunction evaluates to 1 and that an empty disjunction evaluates to 0.

Lemma 7. *Let $\widetilde{\sqcup}$, $\widetilde{\sqcap}$, $D_{B\text{-}STRUCT}{}'$, and $\alpha' : D_{2\text{-}STRUCT} \rightarrow D_{B\text{-}STRUCT}{}'$ be defined analogously to their corresponding definitions in Section 2, substituting traditional embedding with loose embedding. The abstraction function α' is monotonic.*

Proof. By the definitions of α and α', and since $\widetilde{\sqsubseteq}$ generalizes $\sqsubseteq$, we have for any $XS \in D_{2\text{-}STRUCT}$

$$\alpha'(XS) = \widetilde{\bigsqcup}_{S \in XS} \{\beta(S)\} = \widetilde{\bigsqcup}_{S' \in \alpha(XS)} \{S'\} \subseteq \alpha(XS) \ .$$

Let $XS_1, XS_2 \in D_{2\text{-}STRUCT}$ be such that $XS_1 \subseteq XS_2$, and let $S_1 \in \alpha'(XS_1)$. We have that $S_1 \in \alpha(XS_1)$, therefore by monotonicity of α and by Definition 4 there exists $S_2 \in \alpha(XS_2)$ such that $S_1 \sqsubseteq S_2$, implying $S_1 \mathrel{\widetilde{\sqsubseteq}} S_2$. If $S_2 \in \alpha'(XS_2)$ then there is nothing to show. Otherwise, by the definition of α' and loose join there exists some $S_2' \in \alpha'(XS_2)$ such that $S_2 \mathrel{\widetilde{\sqsubseteq}} S_2'$, then by transitivity of loose embedding we have $S_1 \mathrel{\widetilde{\sqsubseteq}} S_2'$. Hence, by (loose extended) Definition 4 we have $\alpha'(XS_1) \mathrel{\widetilde{\sqsubseteq}} \alpha'(XS_2)$. □

5.4 Local Soundness

The proposed extensions 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 $\exists u.eq(u,u) \wedge \phi$, this way assuring that any predicate which is existentially quantified over a summary node is "lowered" to $\frac{1}{2}$, accounting 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 $\forall u.\neg eq(u,u) \vee P$, this way assuring that any universally quantified predicate is "raised" to $\frac{1}{2}$ for *any* summary node, accounting for the absence of corresponding nodes in some concrete settings which could cause the 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 this predicate's values along any sequence of (zero or more) summary nodes between v_1 and v_2. As opposed to Definition 5, here we cannot consider the set of non-image summaries, 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.

More formally, we interpret $p(u_1, u_2)$ as

$$p(u_1,u_2) \vee \bigvee_{v_1,\ldots,v_k \in U} \Bigg(p(u_1,v_1) \\ \wedge \Bigg(\bigwedge_{1 \le i \le k-1} \neg eq(v_i,v_i) \wedge p(v_i,v_{i+1}) \Bigg) \wedge \neg eq(v_k,v_k) \wedge p(v_k,u_2) \Bigg) \ .$$

This implies that only summary nodes ($eq(v, v) = \frac{1}{2}$) can affect the interpretation when considered as intermediate individuals for binary transitive closure, and that such an effect is constrained to lift the interpretation up to the non-definite ($\frac{1}{2}$) value. In turn, this guarantees that the interpretation of a binary predicate retains Kleene ordering across loose embedding functions, ensuring a conservative approximation.

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. It is worth mentioning that loose embedding also deflates some of our structure-based refinement operators, like the two single-noded structures in Fig. 6(a) which are now embedded in their respective general case counterparts.

6 Experimental Results

Table 2 presents analysis statistics for a set of five small Java programs, manipulating singly- or doubly-linked lists, executed on a 1.6 GHz Pentium-M, 512 Mb machine running Linux. This benchmark, along with approximate analysis statistics using the TVLA reference implementation on similar hardware, were adopted from [2]. 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 which deletes an arbitrary element from a singly-linked list). Although this kind of an improvement was expected—our analyzer is restricted by construction and therefore better tweaked for performance—the actual speed-up factor is quite encouraging. The time spent on abstraction 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

	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 top number of structures for a CFG node, analysis total and refinement times, and peak memory consumption). Time is in miliseconds, memory is in kilobytes

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—one example is the use of dense vector representation of node (unary) and edge (binary) attributes—and could probably be improved significantly in the future.

Second, the use 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 bear the greatest gain in state set deflation, flattening peak number of states from 48 to 20 and from 9.9 to 4.4 on average, consequently yielding a significant drop in the running time of the analysis. 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—compared to those of a traditional (strict) analysis. We consider this a nice practical outcome, which supports our view of the problem with strict embedding abstraction.

7 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, powerset heap abstractions were compressed into singleton sets [5] or partially disjunctive sets [8] by means of merging (joining) predicates values and allowing individuals to represent no concrete nodes, or merging structures bearing isomorphic sets of individuals, respectively. Our loose embedding approach seems to resemble the former to some extent, as both allow certain nodes to represent no 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 notion of embedding relation—rather than overloading predicates or joining structures—our approach has the advantage of not inducing imprecision on surviving representative abstract states.

Other approaches deviate from 3-valued canonical abstraction and examine the use of predicate abstraction for analyzing shape properties [9, 4]. While such approaches were shown to yield precise and descriptive results, their contribution to scalability of shape problems is in question due to the large number of predicates required for sufficient precision.

As far as we know, our work is the first attempt to deploy practical structure-based abstraction refinement, and may serve as a point of reference for future efforts.

8 Conclusion

We described a new implementation of a 3-valued logic based shape analysis tool which uses an effective structure-based approach for abstraction refinement, and deflates abstract state sets using an alternate definition of abstract states 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 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 hand.

Acknowledgments

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

References

[1] G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Intersecting heap abstractions with applications to compile-time memory management. Technical Report TR-2005-04-135520, Tel-Aviv University, Apr. 2005. Available at http://www.cs.tau.ac.il/~rumster/TR-2005-04-135520.pdf.

[2] 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.

[3] 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.

[4] 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.

[5] T. Lev-Ami. TVLA: A framework for kleene logic based static analysis. Master's thesis, Tel-Aviv University, May 2000.

[6] 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.

[7] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS)*, pages 280–301, 2000.

[8] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Static Analysis Symposium (SAS)*, pages 265–279, 2004.

[9] 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.

[10] T. W. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *ESOP*, pages 380–398, 2003.

[11] 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.

[12] 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.