Parametric Shape Analysis via 3-Valued Logic

Mooly Sagiv* Tel-Aviv Univ.
Thomas Reps† Univ. of Wisconsin
Reinhard Wilhelm‡ Univ. des Saarlandes

Abstract

We present a family of abstract-interpretation algorithms that are capable of determining "shape invariants" of programs that perform destructive updating on dynamically allocated storage. The main idea is to represent the stores that can possibly arise during execution using three-valued logical structures.

Questions about properties of stores can be answered by evaluating predicate-logic formulae using Kleene's semantics of three-valued logic:

- If a formula evaluates to *true*, then the formula holds in every store represented by the three-valued structure.
- If a formula evaluates to *false*, then the formula does not hold in any store represented by the three-valued structure.
- If a formula evaluates to *unknown*, then we do not know if this formula always holds, never holds, or sometimes holds and sometimes does not hold in the stores represented by the three-valued structure.

Three-valued logical structures are thus a conservative representation of memory stores.

The approach described is a *parametric* framework: It provides the basis for generating a family of shape-analysis algorithms by varying the vocabulary used in the three-valued logic.

1 Introduction

Data structures built using pointers can be characterized by invariants describing their "shape" at stable states, i.e., in between operations on them. These invariants are usually not preserved by the execution of individual program statements, and it is challenging to prove that invariants are reestablished once a sequence of operations is finished [9]. In the past two decades, many "shape-analysis" algorithms have been developed that can automatically identify shape invariants in some programs that manipulate heap-allocated storage [11, 12, 15, 10, 2, 21, 1, 16, 22, 19]. A common feature of these algorithms is that they represent heap cells by "shape-nodes" and sets of "indistinguishable" run-time locations by a single shape-node, often called a *summary-node* [2]. One way of looking at these algorithms is that "shape graphs" are indirect representations of store invariants.

1.1 Main Results

This paper presents a *parametric* framework for shape analysis. Different instantiations of the framework allow the usage patterns of different kinds of data structures in a program to be observed, or allow the usage patterns of data structures to be observed with different levels of precision and efficiency. The ideal is to have a fully automatic method—a `yacc` for shape analysis, so to speak. The "designer" of a shape-analysis algorithm would supply *only* the specification, and the shape-analysis algorithm would be created automatically from this specification. This can be achieved by means of the methods presented in this paper.

Moreover, the framework allows us to create algorithms that are more precise than the above-cited algorithms. In particular, by tracking which run-time locations are *reachable* from which program variables, it is often possible to determine precise shape information for programs that manipulate several (possibly cyclic) data structures. Other static-analysis techniques (including ones that are not based on shape graphs [14, 6, 8, 4, 5]) yield very imprecise information on these programs.

1.1.1 The Use of Logic for Shape Analysis

In our shape-analysis framework, predicate-logic formulae play many roles: expressing both the concrete and abstract semantics of the programming language, expressing properties of store elements (e.g., may-aliases, must-aliases), and expressing properties of stores (e.g., data-structure invariants). For instance, the predicate $x(v)$ expresses whether pointer variable `x` points to heap cell v; the binary predicate $n(v_1, v_2)$ express whether the `n`-component of heap cell v_1 points to heap cell v_2; to specify the effect of the statement "`x = x->n`" on variable `x` (part of the concrete semantics), we write the formula

$$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v). \tag{1}$$

This indicates that after this statement, variable `x` points to a heap cell that was formerly pointed to by `x->n`. To express the property "program variables `x` and `y` are not may-aliases", we write the formula

$$\forall v : \neg(x(v) \wedge y(v)). \tag{2}$$

1.1.2 Shape Analysis via Three-Valued Logic

We use Kleene's three-valued logic [13] (which has a third truth value that signifies "unknown") to create a shape-analysis algorithm automatically from a specification. Kleene's logic is useful for shape analysis because we only have partial information about summary nodes: For these nodes, predicates may have the value *unknown*. One of the nice properties of Kleene's three-valued logic is that the interpretations of formulae in two-valued and three-valued logic coincide on *true* and *false*. This comes in handy for shape analysis, where we wish to relate the concrete (two-valued) world and the abstract (three-valued) world: The advantage of using logic is that it allows us to make a statement about *both* the concrete

*Supported in part by the U.S.-Israel BSF under grant 96-00337.
Address: Dept. of Comp. Sci.; Tel-Aviv Univ.; Tel-Aviv 69978; Israel. E-mail: sagiv@math.tau.ac.il.

†Supported in part by the NSF under grants CCR-9625667 and CCR-9619219, by the U.S.-Israel BSF under grant 96-00337, by grants from Rockwell and IBM, and by a Vilas Associate Award from the Univ. of Wisconsin.
Address: Comp. Sci. Dept.; Univ. of Wisconsin; 1210 W. Dayton St.; Madison, WI 53706; USA. E-mail: reps@cs.wisc.edu.

‡Supported in part by a DAAD-NSF Collaborative Research Grant.
Address: Fachbereich 14 Informatik; Univ. des Saarlandes; 66123 Saarbrücken; Germany. E-mail: wilhelm@cs.uni-sb.de.

To appear in the Proceedings of the Twenty-Sixth ACM Symposium on Principles of Programming Languages, San Antonio, TX, Jan., 1999.

and abstract worlds via the *same* formula—the same syntactic expression can be interpreted either as statement about the two-valued world or the three-valued world.

In this paper, shape graphs are represented as "three-valued logical structures" that provide truth values for every formula. Therefore, by evaluating formulae, one obtains simple algorithms for: (i) executing statements abstractly, and (ii) (conservatively) extracting store properties from a shape graph. For example, formula (2) evaluates to *true* for an abstract store in which x and y do not point to the same shape-node. In this case, we know that x and y cannot be aliases. Formula (2) evaluates to *false* for an abstract store in which x and y point to the same non-summary node. In this case, we know that x and y are aliases. However, the formula can evaluate to *unknown* when both `x` and `y` point to a summary-node. In this case, the analysis does not know if x and y can be aliases.

In Sections 2 and 4, we show how these mechanisms can be exploited to create a parametric framework for shape-analysis. This technique suffices to explain the algorithms of [11, 10, 2, 21].

1.1.3 Materialization of New Nodes from Summary Nodes

One of the magical aspects of [19] is "materialization", in which a transfer function splits a summary-node into two separate nodes. (This operation is also discussed in [2, 16].) This turns out to be important for maintaining accuracy in the analysis of loops that advance pointers through data structures. The parametric framework provides insight into the workings of materialization. It shows that the essence of materialization involves a step (called *focus*, discussed in Section 5.1) that forces the values of certain formulae from *unknown* to *true* or *false*. This has the effect of converting a shape graph into one with finer distinctions.

In [19], it was observed that node materialization is complicated because various kinds of shape-graph properties are interdependent. For instance, the connections between heap cells constrain the sets of potential aliases, and vice versa. In this paper, we introduce a mechanism for expressing (three-valued) constraints on shape graphs, which we use to capture such dependences between properties.

1.2 Limitations

The results reported in the paper are limited in the following ways:

- The framework creates intraprocedural shape-analysis algorithms, not interprocedural ones. Methods for handling procedures are presented in [2, 1, 19]. Because these are instances of the framework, their methods for handling procedures should generalize to the parametric case.
- The number of possible shape-nodes that may arise during abstract interpretation is potentially exponential in the size of the specification. We do not know how severe this problem is in practice. However, it is possible to define a widening operator that converts a shape graph into a more compact, but possibly less precise, shape graph by collapsing more nodes into summary nodes. This can be used to make a shape-analysis algorithm polynomial, at the cost of making the results less accurate.
- The number of shape graphs may be quite large (as in [11, 10]). This problem was avoided in [15, 2, 16, 19] by keeping a single merged shape graph at every point. This measure has not been employed in this paper in order to simplify the presentation.

(a)

```
/* list.h */
typedef struct node {
  struct node *n;
  int data;
} *List;
```

(b)

```
/* reverse.c */
#include ''list.h''
List reverse(List x) {
  List y, t;
  assert(acyclic_list(x));
  y = NULL;
  while (x != NULL) {
    t = y;
    y = x;
    x = x->n;
    y->n = t;
  }
  return y;
}
```

Figure 1: (a) Declaration of a linked-list data type in C. (b) A C function that uses destructive updating to reverse the list pointed to by parameter `x`.

1.3 Organization of the Paper

We explain our work by presenting two versions of the shape-analysis framework. The first version is used to introduce many of the key ideas, but in a simplified setting: Section 2 provides an overview of the simplified version and presents an example of it in action; Section 4 gives the technical details. Section 3 presents technical details of how three-valued logic is used to define abstractions of concrete stores (which is needed for Section 4 and subsequent sections). Section 5 defines the more elaborate version of the shape-analysis framework. Due to space constraints, some aspects of the abstract semantics are omitted (see [18]). Section 6 contains a short account of related work.

2 An Overview of the Parametric Framework

Figure 1(a) shows the declaration of a linked-list data type in C, and Figure 1(b) shows a C program that reverses a list via destructive updating. The analysis of the shapes of the data structures that arise at the different points in the `reverse` program will serve as the subject of the examples given in the remainder of the paper. The `reverse` program allows us to demonstrate many aspects of the shape-analysis framework in a nontrivial, but still relatively digestible, fashion.

2.1 Representing Stores via Three-Valued Structures

In Section 1, we couched the discussion in terms of shape-graphs for the convenience of readers who are familiar with previous work. Formally, we do not work with shape-graphs; instead, the abstractions of stores will be what logicians call *three-valued logical structures*, denoted by $\langle U, \iota \rangle$. There is a *vocabulary* of predicate symbols (with given arities); each logical structure has a universe of *individuals* U, and ι maps each possible tuple $p(u_1, \ldots, u_k)$ of an arity-k predicate symbol p, where $u_i \in U$, to the value 0, 1, or 1/2, (i.e., *false*, *true*, and *unknown*, respectively). Logical structures are used to provide a uniform representation of stores: Individuals represent abstractions of memory locations; pointers from the stack into the heap are represented by unary "pointed-to-by-variable-x" predicates; and pointer-valued fields of data structures are represented by binary "pointer-component-points-to" predicates.

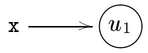
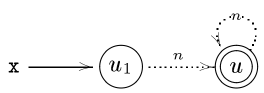
<table>
<tr><th>S</th><th>Structure</th><th>Graphical Representation</th></tr>
<tr><td>S_0</td><td>unary predicates:<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>sm</th><th>is</th></tr></table>binary predicates:<table><tr><th>n</th></tr></table></td><td></td></tr>
<tr><td>S_1</td><td>unary predicates:<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>sm</th><th>is</th></tr><tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>binary predicates:<table><tr><th>n</th><th>u_1</th></tr><tr><td>u_1</td><td>0</td></tr></table></td><td>
</td></tr>
<tr><td>S_2</td><td>unary predicates:<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>sm</th><th>is</th></tr><tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u</td><td>0</td><td>0</td><td>0</td><td>1/2</td><td>0</td></tr></table>binary predicates:<table><tr><th>n</th><th>u_1</th><th>u</th></tr><tr><td>u_1</td><td>0</td><td>1/2</td></tr><tr><td>u</td><td>0</td><td>1/2</td></tr></table></td><td>
</td></tr>
</table>

Figure 2: The three-valued logical structures that describe all possible acyclic inputs to `reverse`.

Assuming that `reverse` is invoked on acyclic lists, the three-valued structures that describe all possible inputs to `reverse` are shown in Figure 2. The following graphical notation is used for three-valued logical structures: Individuals of the universe are represented by circles with names inside. Summary nodes (i.e., nodes for which the value of predicate sm is 1/2) are represented by double circles. Other unary predicates with value 1 (1/2) and binary pointer-component-points-to predicates are represented by solid (dotted) arrows. Thus, in structure S_2, pointer variable `x` points to element u_1, whose `n` field may point to a location represented by element u. u is a summary node, i.e., it may represent more than one location. Possibly there is an `n` field in one of these locations that points to another location represented by u.

S_2 corresponds to stores in which program variable `x` points to an acyclic list of two or more elements:

- The abstract element u_1 represents the head of the list, and u represents all the tail elements.
- The unary predicates x, y, and t are used to characterize the list elements pointed to by program variables `x`, `y`, and `t`, respectively.
- The unary predicate sm indicates whether abstract elements are "summary elements", i.e., represent more than one concrete list element in a given store. Thus, $sm(u_1) = 0$ because u_1 represents a unique list element, the list head. In contrast, $sm(u) = 1/2$, because u represents a single list element when the input list has exactly two elements, and more than one list element when the input list is of length three or more.
- The unary predicate is is explained in Section 2.2.
- The binary predicate n represents the `n` fields of list elements. The value of $n(u_1, u)$ is 1/2 because there are list elements represented by u that are not immediate `n`–successors of u_1.

The structures S_0 and S_1 represent the simpler cases of lists of length zero and one, respectively.

2.2 Conservative Extraction of Store Properties

Three-valued structures offer a systematic way to answer questions about properties of stores:

Observation 2.1 [Property-Extraction Principle]. *Questions about properties of stores can be answered by evaluating formulae using Kleene's semantics of three-valued logic:*

- *If a formula evaluates to 1, then the formula holds in every store represented by the three-valued structure.*
- *If a formula evaluates to 0, then the formula never holds in any store represented by the three-valued structure.*
- *If a formula evaluates to 1/2, then we do not know if this formula always holds, never holds, or sometimes holds and sometimes does not hold.*

□

In Section 3.3, we give the *Embedding Theorem* (Theorem 3.7), which states that the three-valued Kleene interpretation in S of every formula is consistent with the formula's two-valued interpretation in every concrete store that S represents.

Now consider the formula

$$\varphi(v) \stackrel{\text{def}}{=} \exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2, \qquad (3)$$

which expresses the property "Do two or more different cells point to v?" Formula $\varphi(v)$ evaluates to 1/2 in S_2 for $v \mapsto u$, $v_1 \mapsto u$, and $v_2 \mapsto u_1$, because $n(u, u) \wedge n(u_1, u) \wedge u \neq u_1 = 1/2 \wedge 1/2 \wedge 1$, which equals 1/2. The intuition is that because the values of $n(u, u)$ and $n(u_1, u)$ are unknown, we do not know whether or not two different cells point to u.

This uncertainty implies that the tail of the list pointed to by `x` *might* be shared (and the list could be cyclic, as well). In fact, neither of these conditions ever holds in the concrete stores that arise in the `reverse` program.

To avoid this imprecision, our abstract structures have an extra "instrumentation predicate", $is(v)$, that represents the truth values of formula (3) for the elements of concrete structures that v represents. In particular, $is(u) = 0$ in S_2. This fact implies that S_2 can only represent acyclic, unshared lists *even though formula (3) evaluates to* 1/2 *on* u.

The preceding discussion illustrates the following principle:

Observation 2.2 [Instrumentation Principle]. *Suppose S is a three-valued structure that represents concrete store $S^\natural$. By explicitly "storing" in S the values that a formula φ has in $S^\natural$, we can maintain finer distinctions in S than can be obtained by evaluating φ in S.* □

2.3 Simple Abstract Interpretation of Program Statements

Our main tool for expressing the semantics of program statements is based on the Property-Extraction Principle:

Observation 2.3 [Expressing Semantics of Statements via Logical Formulae]. *Suppose a structure S represents a set of stores that arise before statement st. A structure that represents the corresponding set of stores that arise after st can be obtained by extracting a suitable collection of properties from S (i.e., by evaluating a suitable collection of formulae that capture the semantics of st).* □

Figure 3 illustrates the first two iterations of an abstract interpretation of `reverse` on the structure S_2 from Figure 2. The value of a predicate $p(v)$ after a statement executes is obtained by evaluating a predicate-update formula $p'(v)$. The appropriate predicate-update formulae for each statement are shown in the second column of Figure 3. Figure 3 lists a predicate-update formula $p'(v)$ only if predicate p is affected

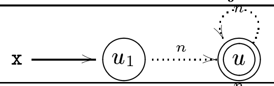
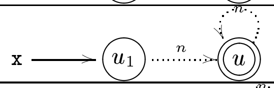
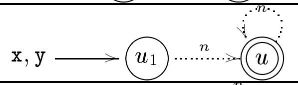
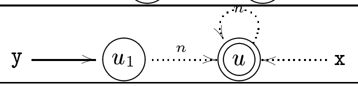
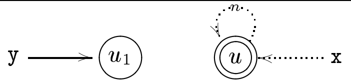
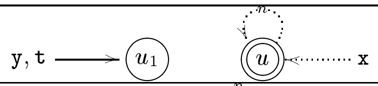
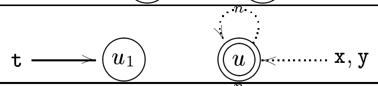
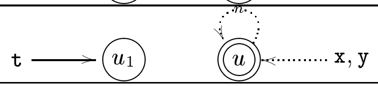
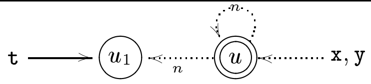
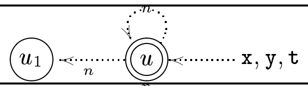
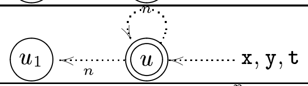
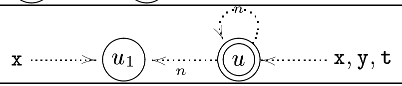
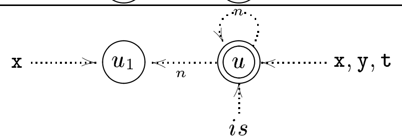
statement	formula	structure that arises just after statement
st_1: `y = NULL;`	$y'(v) = \mathbf{0}$	 S_3
st_2: `t = y;`	$t'(v) = y(v)$	 S_4
st_3: `y = x;`	$y'(v) = x(v)$	 S_5
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$	 S_6
st_5: `y->n = t;`	$n'(v_1, v_2) = (n(v_1, v_2) \wedge \neg y(v_1)) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = \begin{array}{l} is(v) \wedge \exists v_1, v_2 : \left(\begin{array}{l} v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v) \\ \wedge \neg y(v_1) \wedge \neg y(v_2) \end{array} \right) \\ \vee (t(v) \wedge \exists v_1 : n(v_1, v) \wedge \neg y(v_1)) \end{array}$	 S_7
st_2: `t = y;`	$t'(v) = y(v)$	 S_8
st_3: `y = x;`	$y'(v) = x(v)$	 S_9
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$	 S_{10}
st_5: `y->n = t;`	$n'(v_1, v_2) = (n(v_1, v_2) \wedge \neg y(v_1)) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = \begin{array}{l} is(v) \wedge \exists v_1, v_2 : \left(\begin{array}{l} v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v) \\ \wedge \neg y(v_1) \wedge \neg y(v_2) \end{array} \right) \\ \vee (t(v) \wedge \exists v_1 : n(v_1, v) \wedge \neg y(v_1)) \end{array}$	 S_{11}
st_2: `t = y;`	$t'(v) = y(v)$	 S_{12}
st_3: `y = x;`	$y'(v) = x(v)$	 S_{13}
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$	 S_{14}
st_5: `y->n = t;`	$n'(v_1, v_2) = (n(v_1, v_2) \wedge \neg y(v_1)) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = \begin{array}{l} is(v) \wedge \exists v_1, v_2 : \left(\begin{array}{l} v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v) \\ \wedge \neg y(v_1) \wedge \neg y(v_2) \end{array} \right) \\ \vee (t(v) \wedge \exists v_1 : n(v_1, v) \wedge \neg y(v_1)) \end{array}$	 S_{15}

Figure 3: The first three iterations of the abstract interpretation of `reverse` (via the simplified framework described in Section 4). In this example, `reverse` is applied to structure S_2 from Figure 2, which represents lists of length two or more.

by the execution of the statement. The shape-analysis algorithm illustrated in Figure 3 is essentially that of Chase et al. [2].

Unfortunately, there is also bad news: The method described above and illustrated in Figure 3 can be very imprecise. For instance, statement st_4 sets `x` to `x->n`; i.e., it makes `x` point to the next element in the list. In the abstract interpretation, the following things occur:

- In the first abstract execution of st_4, $x'(u)$ is set to 1/2 because $x(u_1) \wedge n(u_1, u) = 1 \wedge 1/2 = 1/2$. In other words, `x` may point to one of the cells represented by the summary node u (see the structure S_6).
- This eventually leads to the situation that occurs after the third abstract execution of st_5, which produces structure S_{15}. Structure S_{15} indicates that "`x`, `y`, and `t` may all point to the same (possibly shared) list".

In Section 5, we show how it is possible to go beyond the simplified approach described above by "materializing" new non-summary nodes from summary nodes as data structures are traversed. As we will see, this allows us to determine the correct shape descriptors for the data structures used in the `reverse` program.

3 Three-Valued Logic and Embedding

This section defines a three-valued first-order logic with equality and transitive closure.

We say that the values 0 and 1 are *definite values* and that 1/2 is an *indefinite value*, and define a partial order $\sqsubseteq$ on truth values to reflect information content: $l_1 \sqsubseteq l_2$ denotes that l_1 has more definite information than l_2:

Definition 3.1 *For $l_1, l_2 \in \{0, 1/2, 1\}$, we define the* **information order** *on truth values as follows: $l_1 \sqsubseteq l_2$ if $l_1 = l_2$ or $l_2 = 1/2$. The symbol $\sqcup$ denotes the least-upper bound operation with respect to $\sqsubseteq$.* □

Kleene's semantics of three-valued logic is monotonic in the information order (see Definition 3.4).

Predicate	Intended Meaning
$x(v)$	Does pointer variable `x` point to element v?
$sm(v)$	Does element v represent more than one concrete element?
$n(v_1, v_2)$	Does the `n` field of v_1 point to v_2?

Table 1: The core predicates that correspond to the `List` data-type declaration from Figure 1(a).

3.1 First-Order Formulae with Transitive Closure

Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be a finite set of predicate symbols. We write first-order formulae over $\mathcal{P}$ using the logical connectives $\wedge$, $\vee$, $\neg$, and the quantifiers $\forall$ and $\exists$. The symbol $=$ denotes the equality predicate. The operator '*TC*' denotes transitive closure on formulae. We also use several shorthand notations: For a binary predicate p, $p^+(v_3, v_4)$ is a shorthand for $(TC\ v_1, v_2 : p(v_1, v_2))(v_3, v_4)$; $\varphi_1 \Rightarrow \varphi_2$ is a shorthand for $(\neg\varphi_1 \vee \varphi_2)$; and $\varphi_1 \Leftrightarrow \varphi_2$ is a shorthand for $(\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$.

Formally, the syntax of first-order formulae with equality and transitive closure is defined as follows:

Definition 3.2 *A* **formula** *over a* **vocabulary** *$\mathcal{P} = \{p_1, \ldots, p_n\}$ is defined inductively, as follows:*

Atomic Formulae *The* **logical-literals 0**, **1**, *and* **1/2** *are atomic formulae with no free variables.*

For every predicate symbol $p \in \mathcal{P}$ of arity k, $p(v_1, \ldots, v_k)$ is an atomic formula with free variables $v_1, \ldots, v_k$.

The formula $(v_1 = v_2)$ is an atomic formula with free variables v_1 and v_2.

Logical Connectives *If φ_1 and φ_2 are formulae whose sets of free variables are V_1 and V_2, respectively, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, and $(\neg\varphi_1)$ are formulae with free variables $V_1 \cup V_2$, $V_1 \cup V_2$, and V_1, respectively.*

Quantifiers *If φ is a formula with free variables $v_1, v_2, \ldots, v_k$, then $(\exists v_1 : \varphi)$ and $(\forall v_1 : \varphi)$ are both formulae with free variables $v_2, v_3, \ldots, v_k$.*

Transitive Closure *If φ is a formula with free variables V such that $v_1, v_2 \in V$ and $v_3, v_4 \notin V$, then $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$ is a formula with free variables $(V - \{v_1, v_2\}) \cup \{v_3, v_4\}$.*

A formula is **closed** *when it has no free variables.* $\square$

In our application, the set of predicates $\mathcal{P}$ is partitioned into two disjoint sets: the "core-predicates", $\mathcal{C}$, and the "instrumentation-predicates", $\mathcal{I}$. The core-predicates are part of the programming-language semantics. In contrast, the instrumentation predicates are introduced in order to improve the precision of the analysis (as described by Observation 2.2).

Example 3.3 Table 1 contains the core-predicates for the `List` data-type declaration from Figure 1(a) and the `reverse` program of Figure 1(b). $\square$

Table 2 lists some interesting instrumentation predicates, and Table 3 lists their defining formulae.

- The sharing predicate is was introduced in [2] and also used in [19] to capture list and tree data structures.
- The reachability-from-`x` predicate r_x was mentioned in [19, p.38]. It drastically improves the precision of shape analysis, even for programs that manipulate simple list and tree data structures, since it keeps separate the abstract representations of data structures that are disjoint in the concrete world.

Pred.	Intended Meaning	Purpose	Ref.
$is(v)$	Do two or more fields of heap elements point to v?	lists and trees	[2], [19]
$r_x(v)$	Is v (transitively) reachable from pointer variable `x`?	separating disjoint data structures	[19]
$r(v)$	Is v reachable from some pointer variable (i.e., is v a non-garbage element)?	compile-time garbage collection	
$c(v)$	Is v on a directed cycle?	ref. counting	[11]
$c_{f.b}(v)$	Does a field-`f` dereference from v, followed by a field-`b` dereference, yield v?	doubly-linked lists	[7], [16]
$c_{b.f}(v)$	Does a field-`b` dereference from v, followed by a field-`f` dereference, yield v?	doubly-linked lists	[7], [16]

Table 2: Examples of instrumentation predicates.

$$\varphi_{is}(v) \stackrel{\text{def}}{=} \exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2 \tag{4}$$

$$\varphi_{r_x}(v) \stackrel{\text{def}}{=} x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v) \tag{5}$$

$$\varphi_r(v) \stackrel{\text{def}}{=} \bigvee_{\mathtt{x} \in PVar} (x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v)) \tag{6}$$

$$\varphi_c(v) \stackrel{\text{def}}{=} n^+(v, v) \tag{7}$$

$$\varphi_{c_{f.b}}(v) \stackrel{\text{def}}{=} \forall v_1, v_2 : f(v, v_1) \wedge b(v_1, v_2) \Rightarrow v_2 = v \tag{8}$$

$$\varphi_{c_{b.f}}(v) \stackrel{\text{def}}{=} \forall v_1, v_2 : b(v, v_1) \wedge f(v_1, v_2) \Rightarrow v_2 = v \tag{9}$$

Table 3: Formulae that define the meaning of the instrumentation predicates listed in Table 2.

- The reachability predicate r identifies non-garbage cells. This is useful for determining when compile-time garbage collection can be performed.
- The cyclicity predicate c was introduced by Jones and Muchnick [11] to aid in determining when reference counting would be sufficient.
- The special cyclicity predicates $c_{f.b}$ and $c_{b.f}$ are used to capture doubly-linked lists, in which forward and backward field dereferences cancel each other. This idea was introduced in [7] and also used in [16].

3.2 Kleene's Three-Valued Semantics

In this section, we define Kleene's three-valued semantics for first-order formulae with transitive closure.

Definition 3.4 *A* **three-valued interpretation** *of the language of formulae over $\mathcal{P}$ is a* **three-valued logical structure** *$S = \langle U^S, \iota^S \rangle$, where U^S is a set of* **individuals** *and ι^S maps each predicate symbol p of arity k to a truth-valued function:*
$\iota^S : \mathcal{P} \to (U^S)^k \to \{0, 1, 1/2\}$.

An **assignment** *Z is a function that maps free variables to individuals (i.e., an assignment has the functionality $Z \colon \{v_1, v_2, \ldots\} \to U^S$). An assignment that is defined on all free variables of a formula φ is called* **complete** *for φ. In the sequel, we assume that every assignment Z that arises in connection with the discussion of some formula φ is complete for φ.*

The **meaning** *of a formula φ, denoted by $[\![\varphi]\!]_3^S(Z)$, yields a truth value in $\{0, 1, 1/2\}$. The meaning of φ is defined inductively as follows:*

Atomic *For a logical-literal* $\mathbf{l} \in \{\mathbf{0}, \mathbf{1}, \mathbf{1/2}\}$, $[\![\mathbf{l}]\!]_3^S(Z) = l$ *(where* $l \in \{0, 1, 1/2\}$*).*

For an atomic formula $p(v_1, \ldots, v_k)$,

$$[\![p(v_1, \ldots, v_k)]\!]_3^S(Z) = \iota^S(p)(Z(v_1), \ldots, Z(v_k))$$

For an atomic formula $(v_1 = v_2)$,

$$[\![v_1 = v_2]\!]_3^S(Z) = \begin{cases} 0 & Z(v_1) \neq Z(v_2) \\ 1 & \begin{array}{c} Z(v_1) = Z(v_2) \\ \wedge\ \iota^S(sm)(Z(v_1)) = 0 \end{array} \\ 1/2 & \text{otherwise} \end{cases}$$

Logical Connectives *For logical formulae* φ_1 *and* φ_2

$$[\![\varphi_1 \wedge \varphi_2]\!]_3^S(Z) = \min([\![\varphi_1]\!]_3^S(Z), [\![\varphi_2]\!]_3^S(Z))$$
$$[\![\varphi_1 \vee \varphi_2]\!]_3^S(Z) = \max([\![\varphi_1]\!]_3^S(Z), [\![\varphi_2]\!]_3^S(Z))$$
$$[\![\neg\varphi_1]\!]_3^S(Z) = 1 - [\![\varphi_1]\!]_3^S(Z)$$

Quantifiers *If* φ *is a logical formula,*

$$[\![\forall v_1 : \varphi]\!]_3^S(Z) = \min_{u \in U^S} [\![\varphi_1]\!]_3^S(Z[v_1 \mapsto u])$$
$$[\![\exists v_1 : \varphi]\!]_3^S(Z) = \max_{u \in U^S} [\![\varphi_1]\!]_3^S(Z[v_1 \mapsto u])$$

Transitive Closure *For* $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$,

$$[\![(TC\ v_1, v_2 : \varphi)(v_3, v_4)]\!]_3^S(Z) = \max_{\substack{u_1, \ldots, u_n \in U, \\ Z(v_3) = u_1, Z(v_4) = u_n}} \min_{i=1}^{n-1} [\![\varphi]\!]_3^S(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

We say that S *and* Z **potentially satisfy** φ *(denoted by* $S, Z \models \varphi$*) if* $[\![\varphi]\!]_3^S(Z) = 1/2$ *or* $[\![\varphi]\!]_3^S(Z) = 1$. *Finally, we write* $S \models \varphi$ *if for every* Z: $S, Z \models \varphi$. □

The only nonstandard part of Definition 3.4 is the meaning of equality (denoted by the symbol '$=$'). The predicate $=$ is defined in terms of the sm predicate and the "identically-equal" relation on individuals (denoted by the symbol '$=$'):[1]

- Non-identical individuals u_1 and u_2 are unequal (i.e., if $u_1 \neq u_2$ then $u_1 \neq u_2$).
- A non-summary individual must be equal to itself (i.e., if $sm(u) = 0$, then $u = u$).
- In all other cases, we throw up our hands and return 1/2.

Three-valued logic retains a number of properties that are familiar from two-valued logic, such as commutativity and associativity of $\wedge$ and $\vee$, distributivity of $\wedge$ over $\vee$ and vice versa, De Morgan laws, etc.

3.3 The Embedding Theorem

In this section, we formulate the Embedding Theorem, which gives us a tool to relate two- and three-valued interpretations. We define the *embedding ordering* on structures as follows:

Definition 3.5 *Let* $S = \langle U^S, \iota^S \rangle$ *and* $S' = \langle U^{S'}, \iota^{S'} \rangle$ *be two structures. Let* $f \colon U^S \to U^{S'}$ *be surjective. We say that* f **embeds** S *in* S' *(denoted by* $S \sqsubseteq^f S'$*) if (i) for every predicate symbol* p *of arity* k *and all* $u_1, \ldots, u_k \in U^S$,

$$\iota^S(p)(u_1, \ldots, u_k) \sqsubseteq \iota^{S'}(p)(f(u_1), \ldots, f(u_k)) \tag{10}$$

and (ii) for all $u' \in U^{S'}$,

$$(|\{u \mid f(u) = u'\}| > 1) \sqsubseteq \iota^{S'}(sm)(u') \tag{11}$$

We say that S **can be embedded in** S' *(denoted by* $S \sqsubseteq S'$*) if there exists a function* f *such that* $S \sqsubseteq^f S'$. □

Note that inequality (10) applies to the summary predicate, sm, as well.

A special kind of embedding is a *tight embedding*, in which information loss is minimized when multiple individuals of S are mapped to the same individual in S':

Definition 3.6 *A structure* $S' = \langle U^{S'}, \iota^{S'} \rangle$ *is a* **tight embedding** *of* $S = \langle U^S, \iota^S \rangle$ *if there exists a surjective function* $t\_embed \colon U^S \to U^{S'}$ *such that, for every* $p \in \mathcal{P} - \{sm\}$ *of arity* k,

$$\iota^{S'}(p)(u'_1, \ldots, u'_k) = \bigsqcup_{t\_embed(u_i) = u'_i, 1 \leq i \leq k} \iota^S(p)(u_1, \ldots, u_k) \tag{12}$$

and for every $u' \in U^{S'}$,

$$\iota^{S'}(sm)(u') = \begin{array}{l}(|\{u | t\_embed(u) = u'\}| > 1) \sqcup \\ \displaystyle\bigsqcup_{t\_embed(u) = u'} \iota^S(sm)(u)\end{array} \tag{13}$$

Because t_embed *is surjective, equations (12) and (13) uniquely determine* S' *(up to isomorphism); therefore, we say that* $S' = t\_embed(S)$. □

It is immediately apparent from Definition 3.6 that the tight embedding of a structure S by a function t_embed possessing properties (12) and (13) embeds S in $t\_embed(S)$, i.e., $S \sqsubseteq^{t\_embed} t\_embed(S)$.

If $f \colon U^S \to U^{S'}$ is a function and $Z \colon Var \to U^S$ is an assignment, $f \circ Z$ denotes the assignment $f \circ Z \colon Var \to U^{S'}$ such that $(f \circ Z)(v) = f(Z(v))$.

We are now ready to state the embedding theorem. Intuitively, it says:

> If $S \sqsubseteq^f S'$, then every piece of information extracted from S' via a formula φ is a conservative approximation of the information extracted from S via φ.

Theorem 3.7 [Embedding Theorem]. *Let* $S = \langle U^S, \iota^S \rangle$ *and* $S' = \langle U^{S'}, \iota^{S'} \rangle$ *be two structures and* $f \colon U^S \to U^{S'}$ *such that* $S \sqsubseteq^f S'$. *Then, for every formula* φ *and complete assignment* Z *for* φ, $[\![\varphi]\!]_3^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S'}(f \circ Z)$. □

3.4 Compatible Structures

We use 3-STRUCT[$\mathcal{P}$] to denote the set of general three-valued structures over vocabulary $\mathcal{P}$, and 2-STRUCT[$\mathcal{P}$] to denote the normal two-valued structures over $\mathcal{P}$. (Note that 2-STRUCT[$\mathcal{P}$] $\subseteq$ 3-STRUCT[$\mathcal{P}$].)

Suppose that P is a C program that operates on the `List` data-type of Figure 1(a), and that $S^\natural \in$ 2-STRUCT[$\mathcal{P}$] is a two-valued structure over the appropriate vocabulary. As described in Table 1, our intention is that $S^\natural$ capture a `List`-valued store in the following manner:

- Each cell in heap-allocated storage corresponds to an individual in $U^{S^\natural}$.
- For every individual u, $\iota^{S^\natural}(x)(u) = 1$ if and only if the heap cell that u represents is pointed to by program variable `x`.
- For every pair of individuals u_1 and u_2, $\iota^{S^\natural}(n)(u_1, u_2) = 1$ if and only if the `n` field of u_1 points to u_2.

[1] Note the typographical distinction between the syntactic symbol for equality, namely '$=$', and the symbol for the "identically-equal" relation on individuals, namely '$=$'.

$$\text{for each } \mathtt{x} \in PVar, \forall v_1, v_2 : x(v_1) \wedge x(v_2) \Rightarrow v_1 = v_2 \quad (14)$$
$$\forall v_1, v_2 : (\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2)) \Rightarrow v_1 = v_2 \quad (15)$$
$$\forall v : (\exists v_1, v_2 : v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v)) \Rightarrow is(v) \quad (16)$$
$$\forall v : \neg(\exists v_1, v_2 : v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v)) \Rightarrow \neg is(v) \quad (17)$$
$$\forall v_2, v : (\exists v_1 : \neg is(v) \wedge v_1 \neq v_2 \wedge n(v_1, v)) \Rightarrow \neg n(v_2, v) \quad (18)$$
$$\forall v_1, v : (\exists v_2 : \neg is(v) \wedge v_1 \neq v_2 \wedge n(v_2, v)) \Rightarrow \neg n(v_1, v) \quad (19)$$
$$\forall v_1, v_2 : (\exists v : \neg is(v) \wedge n(v_1, v) \wedge n(v_2, v)) \Rightarrow v_1 = v_2 \quad (20)$$

Table 4: Compatibility formulae F for structures that represent a store of the `reverse` program, which operates on the `List` data-type declaration from Figure 1(a). The rules below the line are logical consequences of the rules above the line, and are generated systematically from the rules above the line, as explained in Section 5.2.1.

(Similar statements hold for the instrumentation predicates, as indicated in Table 2.)

However, not all structures $S^\natural \in$ 2-STRUCT$[\mathcal{P}]$ represent stores that are compatible with the semantics of C. For example, stores have the property that each pointer variable points to at most one element in heap-allocated storage. Consequently, we are not interested in *all* structures in 2-STRUCT$[\mathcal{P}]$, but only in ones compatible with the semantics of C. Table 4 lists a set of *compatibility formulae* F (or "hygiene conditions") that must be satisfied for a structure to represent a store of a C program that operates on the `List` data-type from Figure 1(a). Formula (14) captures the fact that every program variable points to at most one list element. Formula (15) captures a similar invariant on the `n` fields of `List` structures: Whenever the `n` field of a list element is non-`NULL`, it points to at most one list element.

In addition, for every instrumentation predicate $p \in \mathcal{I}$ defined by a formula $\varphi_p(v_1, \ldots, v_k)$, we generate a compatibility formula of the following form:

$$\forall v_1, \ldots, v_k : \varphi_p(v_1, \ldots, v_k) \Leftrightarrow p(v_1, \ldots, v_k) \quad (21)$$

This is then broken into two formulae of the form:

$$\forall v_1, \ldots, v_k : \varphi_p(v_1, \ldots, v_k) \Rightarrow p(v_1, \ldots, v_k)$$
$$\forall v_1, \ldots, v_k : \neg\varphi_p(v_1, \ldots, v_k) \Rightarrow \neg p(v_1, \ldots, v_k)$$

For instance, for the instrumentation predicate is, we use formula (4) for φ_{is} to generate compatibility formulae (16) and (17).

In the remainder of the paper, 2-CSTRUCT$[\mathcal{P}, F]$ denotes the set of two-valued structures that satisfy a set of compatibility formulae F.

Compatibility constraints for three-valued structures are discussed in Section 5.2.1.

4 A Simple Abstract Semantics

In this section, we formally work out the abstract-interpretation algorithm that was sketched in Section 2.3. In Section 4.1, we define how (a potentially infinite number of) concrete structures can be represented conservatively using a single three-valued structure. In Section 4.2, the meaning functions of the program statements are defined. To guarantee that the analysis of a program containing a loop terminates, we require that the number of potential structures for a given program be finite. For this reason, in Section 4.3 we introduce the set of bounded structures, and show how every three-valued structure can be mapped into a bounded structure. Section 4.4 states the abstract interpretation in terms of a least fixed point of a set of equations.

4.1 The Concrete Stores Represented by a Three-Valued Structure

Definition 4.1 (Concretization of Three-Valued Structures) *For a structure $S \in$ 3-STRUCT$[\mathcal{P}]$, we denote by $\gamma(S)$ the set of two-valued structures that S represents, i.e.,*

$$\gamma(S) = \{S^\natural \mid S^\natural \sqsubseteq S, S^\natural \in \text{2-CSTRUCT}[\mathcal{P}, F]\} \quad (22)$$

□

Example 4.2 The structure S_2 shown in Figure 2 represents lists of length two or more. □

4.2 The Meaning of Program Statements

In this subsection, we present a simple algorithm that, given a program, computes for every point in the program a conservative approximation of the set of concrete structures that arise at that point during execution. (This algorithm is refined in Section 5 to obtain a more precise solution.)

We now formalize the abstract semantics that was discussed in Section 2.3. The main idea is that for every statement st, the new values of every predicate p are defined via a predicate-update formula φ_p^{st} (referred to as p' in Section 2.3).

Definition 4.3 *Let st be a program statement, and for every arity-k predicate p in vocabulary $\mathcal{P}$, let φ_p^{st} be the formula over free variables $v_1, \ldots, v_k$ that defines the new value of p after st. Then the $\mathcal{P}$* **transformer associated with** *st, denoted by $[\![st]\!]$, is defined as follows:*

$$[\![st]\!](S) = \left\langle \begin{array}{l} U^S, \\ \lambda p.\lambda u_1, \ldots, u_k.[\![\varphi_p^{st}]\!]_3^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]) \end{array} \right\rangle$$

□

Example 4.4 Table 5 lists the predicate-update formulae that define the abstract semantics of the five kinds of statements that manipulate data structures defined by the `List` data type given in Figure 1(a). (For the moment, ignore the case for statements of the form `x = malloc()`.) □

Definition 4.3 does not handle statements of the form `x = malloc()` because the universe of S does not change. Instead, for statements of this form, we use the modified definition of $[\![st]\!](S)$ given in Definition 4.5, which first allocates a new individual u_{new}, and then invokes predicate-update formulae in a manner similar to Definition 4.3.

Definition 4.5 *Let $st \equiv$ `x = malloc()` and let $new \notin \mathcal{P}$ be a unary predicate. For every $p \in \mathcal{P}$, let φ_p^{st} be a predicate-update formula over vocabulary $\mathcal{P} \cup \{new\}$. Then the $\mathcal{P}$* **transformer associated with** *$st \equiv$ `x = malloc()`, denoted by $[\![x = malloc()]\!]$, is defined as follows:*

$$[\![\mathtt{x = malloc()}]\!](S) =$$

$$\textbf{let } U' = U^S \cup \{u_{new}\}, \text{ where } u_{new} \text{ is an individual not in } U^S$$
$$\textbf{and } \iota' = \begin{array}{l} \lambda p.\lambda u_1, \ldots, u_k. \\ \begin{cases} 1 & p = new \wedge u_1 = u_{new} \\ 0 & p = new \wedge u_1 \neq u_{new} \\ 1/2 & p \neq new \wedge \bigvee_{1 \leq i \leq k} u_i = u_{new} \\ \iota^S(p)(u_1, \ldots, u_k) & \text{otherwise} \end{cases} \end{array}$$
$$\textbf{in } \left\langle \begin{array}{l} U', \\ \lambda p.\lambda u_1, \ldots, u_k.[\![\varphi_p^{st}]\!]_3^{\langle U', \iota' \rangle}([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]) \end{array} \right\rangle$$

□

In Definition 4.5, ι' is created from ι as follows: (i) $new(u_{new})$ is set to 1, (ii) $new(u_1)$ is set to 0 for all other individuals $u_1 \neq u_{new}$, and (iii) all predicates are set to 1/2 when any

st	φ_p^{st}
`x = NULL`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} \mathbf{0}$ $\varphi_z^{st}(v) \stackrel{\text{def}}{=} z(v)$, for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} n(v_1, v_2)$ $\varphi_{sm}^{st}(v) \stackrel{\text{def}}{=} sm(v)$
`x = t`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} t(v)$ $\varphi_z^{st}(v) \stackrel{\text{def}}{=} z(v)$, for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} n(v_1, v_2)$ $\varphi_{sm}^{st}(v) \stackrel{\text{def}}{=} sm(v)$
`x = t->n`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} \exists v_1 : t(v_1) \wedge n(v_1, v)$ $\varphi_z^{st}(v) \stackrel{\text{def}}{=} z(v)$, for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} n(v_1, v_2)$ $\varphi_{sm}^{st}(v) \stackrel{\text{def}}{=} sm(v)$
`x->n = t`	$\varphi_z^{st}(v) \stackrel{\text{def}}{=} z(v)$, for each $\mathtt{z} \in PVar$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} \begin{array}{l} (n(v_1, v_2) \wedge \neg x(v_1)) \\ \vee\ (x(v_1) \wedge t(v_2)) \end{array}$ $\varphi_{sm}^{st}(v) \stackrel{\text{def}}{=} sm(v)$
`x = malloc()`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} new(v)$ $\varphi_z^{st}(v) \stackrel{\text{def}}{=} \begin{array}{l} z(v) \wedge \neg new(v), \\ \text{for each } \mathtt{z} \in (PVar - \{\mathtt{x}\}) \end{array}$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} \begin{array}{l} n(v_1, v_2) \\ \wedge\ \neg new(v_1) \wedge \neg new(v_2) \end{array}$ $\varphi_{sm}^{st}(v) \stackrel{\text{def}}{=} sm(v) \wedge \neg new(v)$

Table 5: Predicate-update formulae for the core predicates for `List` and `reverse`.

argument is u_{new}. The predicate-update operation in Definition 4.5 is very similar to the one in Definition 4.3 after ι' has been set. (Note that the p in "$\iota' = \lambda p. \ldots$" ranges over $\mathcal{P} \cup \{new\}$, whereas the p in "$\lambda p. \ldots$" appearing in the last line of Definition 4.5 ranges over $\mathcal{P}$.)

The Embedding Theorem immediately implies that the three-valued interpretation is conservative with respect to every store that can possibly occur at run-time.

The above two definitions are not the complete story. In the case of the instrumentation predicates, the statements need to maintain "correct instrumentation". This is formally defined as follows:

Definition 4.6 *A predicate-update formula* φ_p^{st} **maintains a correct instrumentation for predicate** $p \in \mathcal{I}$ *if, for all* $S^\natural \in$ *2-CSTRUCT*$[\mathcal{P}, F]$ *and for all* Z,

$$[\![\varphi_p^{st}]\!]_3^{S^\natural}(Z) = [\![\varphi_p]\!]_3^{[\![st]\!](S^\natural)}(Z). \tag{23}$$

□

Example 4.7 Table 6 gives the definitions of the predicate-update formulae for the instrumentation predicate is. It is not hard to see that, for each kind of assignment statement, equation (23) holds. □

Henceforth, when discussing the general case (i.e., the parametric framework), we assume that all predicate-update formulae maintain correct instrumentations.

4.3 Bounded Structures

To guarantee that shape analysis terminates for a program that contains a loop, we require that the number of potential structures for a given program be finite. Toward this end, we make the following definition:

st	φ_{is}^{st}
`x = NULL`	$\varphi_{is}^{st}(v) \stackrel{\text{def}}{=} is(v)$
`x = t`	$\varphi_{is}^{st}(v) \stackrel{\text{def}}{=} is(v)$
`x = t->n`	$\varphi_{is}^{st}(v) \stackrel{\text{def}}{=} is(v)$
`x->n = t`	$\varphi_{is}^{st}(v) \stackrel{\text{def}}{=} \begin{array}{l} \left(\begin{array}{l} is(v) \wedge \exists v_1, v_2 : v_1 \neq v_2 \\ \wedge\ n(v_1, v) \wedge n(v_2, v) \\ \wedge\ \neg x(v_1) \wedge \neg x(v_2) \end{array}\right) \\ \vee\ (t(v) \wedge \exists v_1 : n(v_1, v) \wedge \neg x(v_1)) \end{array}$
`x = malloc()`	$\varphi_{is}^{st}(v) \stackrel{\text{def}}{=} is(v) \wedge \neg new(v)$

Table 6: Predicate-update formulae for the instrumentation predicate is.

Definition 4.8 *A* **bounded structure** *over vocabulary* $\mathcal{P}$ *is a structure* $S = \langle U^S, \iota^S \rangle$ *such that for every* $u_1, u_2 \in U^S$, *where* $u_1 \neq u_2$, *there exists a unary predicate symbol* $p \in \mathcal{P}$ *such that (i)* $\iota^S(p)(u_1) \neq 1/2$, *(ii)* $\iota^S(p)(u_2) \neq 1/2$, *and (iii)* $\iota^S(p)(u_1) \neq \iota^S(p)(u_2)$.

In the sequel, B-STRUCT$[\mathcal{P}]$ *denotes the set of such structures.* □

There are two consequences of Definition 4.8:

- For every fixed set of predicate symbols $\mathcal{P}$ containing unary predicate symbols $\mathcal{A} \subseteq \mathcal{P}$, there is an upper bound on the size of structures $S \in$ B-STRUCT$[\mathcal{P}]$, i.e., $|U^S| \leq 2^{|\mathcal{A}|}$.
- The embedding of any structure into a bounded structure S is unique.

Example 4.9 Consider the class of bounded structures associated with the `List` data-type declaration from Figure 1(a). Here the predicate symbols are $\mathcal{C} = \{sm, n\} \cup \{x \mid \mathtt{x} \in PVar\}$ and $\mathcal{I} = \{is\}$.[2]

For the `reverse` program from Figure 1(b), the program variables are `x`, `y`, and `t`, yielding unary core predicates x, y, and t; the other unary predicates are is and sm. Therefore, the maximum number of individuals in a structure is $2^5 = 32$; however, a consequence of equation (13) is that sm cannot have the value 1, and thus the maximum number of individuals in a structure is really only 16. On the other hand, Figure 3 shows that each structure that arises in the analysis of `reverse` has at most two individuals. □

One way to obtain a bounded structure is to map individuals into abstract individuals named by the definite values of the unary predicate symbols. That is, to embed unbounded-size structures into bounded-size ones, we exploit the following **abstraction principle**, in which the mapping is controlled by a fixed set of unary "abstraction predicates"—the unary predicates of the vocabulary:

> Individuals are partitioned into equivalence classes according to their sets of unary-predicate values. Every structure $S^\natural$ is then represented (conservatively) by a condensed structure in which each individual of S represents an equivalence class of $S^\natural$.

This is formalized in the following definition:

[2] The predicate sm has a slightly different status than the other core predicates. It captures the essence of "summary-nodes", and thus has has a fixed meaning in all concrete structures, namely, $sm(u) = 0$ for all $u \in U^S$. Including sm in the concrete structures allows us to work with the same vocabularies at the concrete and abstract levels.

Definition 4.10 *The* **canonical abstraction** *of a structure S, denoted by $t\_embed_c(S)$, is the tight embedding induced by the following mapping:*

$t\_embed_c(u) = u_{\{p \in \mathcal{A}-\{sm\} \mid \iota^S(p)(u)=1\}, \{p \in \mathcal{A}-\{sm\} \mid \iota^S(p)(u)=0\}}$.
□

Note that t_embed_c can be applied to any three-valued structure, not just two-valued structures, and that t_embed_c is idempotent (i.e., $t\_embed_c(t\_embed_c(S)) = t\_embed_c(S)$).

The name "$u_{\{p \in \mathcal{A}-\{sm\} \mid \iota^S(p)(u)=1\}, \{p \in \mathcal{A}-\{sm\} \mid \iota^S(p)(u)=0\}}$" is known as the **canonical name** of individual u.

Example 4.11 In structure S_2 from Figure 2, the canonical name of individual u_1 is $u_{\{x\},\{y,t,is\}}$, and the canonical name of u is $u_{\emptyset,\{x,y,t,is\}}$. In structure S_5, which arises after the first abstract interpretation of statement st_3 in Figure 3, the canonical name of u_1 is $u_{\{x,y\},\{t,is\}}$, and the canonical name of u is $u_{\emptyset,\{x,y,t,is\}}$. □

It is straightforward to generalize Definition 4.10 to use just a subset of the unary predicate symbols, rather than all of the unary predicate symbols $\mathcal{A} \subseteq \mathcal{P}$. This alternative yields bounded structures that have a smaller number of individuals, but may decrease the precision of the shape-analysis algorithm. For instance, Definition 4.10 is a generalization of the abstraction function used in [19].[3] The only abstraction predicates used in [19] are the "pointed-to-by-`x`" predicates; the predicate is is used only as an instrumentation predicate in [19], but not as an abstraction predicate (i.e., is does not contribute to the canonical name of an individual in [19]). Consequently, the algorithm from [19] loses precision for stores that contain both shared and unshared heap cells that are not directly pointed to by any variable. Adopting is as an additional abstraction predicate improves the accuracy of shape analysis: In this case, shared heap cells and unshared heap cells are represented by abstract individuals that have different canonical names.

4.4 The Shape-Analysis Algorithm

In this section, we define the actual shape-analysis algorithm.

Definition 4.12 *For structure sets $XS_1, XS_1 \subseteq$ 3-STRUCT$[\mathcal{P}]$, we define: $XS_1 \sqsubseteq XS_2 \Leftrightarrow \forall S_1 \in XS_2 : \exists S_2 \in XS_2 : S_1 \sqsubseteq S_2$.*
□

The shape-analysis algorithm itself is an iterative procedure that computes a set of structures, $StructSet[v]$, for each vertex v of control-flow graph G, as a least fixed point of the following system of equations over the variables $StructSet[v]$:

$$StructSet[v] = \begin{cases} \bigcup\limits_{w \to v \in G} \{t\_embed_c[\![st(w)]\!](S) \mid S \in StructSet[w]\} & \\ & \text{if } v \neq start \\ \{\langle \emptyset, \lambda p.\lambda u_1, \ldots, u_k.1/2 \rangle\} & \text{if } v = start \end{cases}$$

The iteration starts from the initial assignment $StructSet[v] = \emptyset$ for each control-flow-graph vertex v. Because of the t_embed_c operation, it is possible to check efficiently if two structures are isomorphic.

5 Improved Abstract Semantics

In this section, we formulate the improved abstract interpretation referred to in Section 2. This analysis recovers precise shape information for many list-manipulation programs, including ones that manipulate cyclic lists.

[3] The shape-analysis algorithm presented in [19] is described in terms of *Storage Shape Graphs* (SSGs), not bounded structures. Our comparison is couched in terms of the terminology of the present paper.

$$\begin{tikzcd} \{S_5\} \arrow[r, "focus"] \arrow[d, "{[\![st_3]\!]}"'] & \{S_{5,f,0}, S_{5,f,1}, S_{5,f,2}\} \arrow[d, "{[\![st_3]\!]}"] \\ \{S_6\} \sqsupseteq \{S_{6,0}, S_{6,1}, S_{6,2}\} & \{S_{5,o,0}, S_{5,o,1}, S_{5,o,2}\} \arrow[l, "coerce"] \end{tikzcd}$$

Figure 4: One- vs. three-stage abstract semantics of statement st_3. The operation $[\![st]\!]$ was already defined in Section 4. The *focus* and the *coerce* operations are introduced in Sections 5.1 and 5.2, respectively. (This example will be discussed in further detail in Sections 5.1 and 5.2.)

In contrast to the abstract meaning function for a statement st given in Definition 4.3, in this section we decompose the transformer for st into a composition of three functions, as depicted in Figure 4 and explained below:

1. The operation *focus*, defined in Section 5.1, refines three-valued structures such that the formulae that define the meaning of st evaluate to definite values. The *focus* operation thus brings these formulae "into focus".
2. The transformer $[\![st]\!]$, defined in Section 4, is then applied (see Definitions 4.3 and 4.5).
3. The operation *coerce*, defined in Section 5.2, converts a three-valued structure into a more precise three-valued structure by removing incompatibilities. In contrast to the other two operations, *coerce* does not depend on the particular statement st; it can be applied at any step (e.g., right after *focus* and before $[\![st]\!]$) and may improve precision.

It is worthwhile noting that both *focus* and *coerce* are *semantic-reduction* operations (originally defined in [3]). That is, they convert a set of three-valued structures into a more precise set of structures that describe the same set of stores. This property, together with the correctness of the structure transformer $[\![st]\!]$, guarantees that the overall three-stage semantics is correct.

5.1 Bringing Formulae Into Focus

To improve the precision of the simple abstract semantics of Section 4 we define an operation, called *focus*, that forces a given formula φ to a definite value.

5.1.1 The Focus Operation

First, we define an auxiliary operation, *maximal*, that returns the set of maximal structures in a given set of structures:

Definition 5.1 *For a set of structures $XS \subseteq$ 3-STRUCT$[\mathcal{P}]$,*

$$maximal(XS) \stackrel{\text{def}}{=} XS - \{X \in XS \mid \exists X' \in XS : X \sqsubseteq X' \text{ and } X' \not\sqsubseteq X\}$$
□

Definition 5.2 *Given a formula φ, the operation $focus_\varphi$ yields the (potentially infinite) set of structures in which φ evaluates to a definite value, i.e.,*

$$focus_\varphi(S) = maximal\left(\left\{ S' \;\middle|\; \begin{array}{l} S' \in \textit{3-STRUCT}[\mathcal{P}] \\ S' \sqsubseteq S \\ \textit{for all } Z : [\![\varphi]\!]_3^{S'}(Z) \neq 1/2 \end{array} \right\}\right)$$
□

Example 5.3 The upper part of Figure 5 illustrates the application of *focus* to the formula $\varphi_x^{st_4}(v)$ and the structure S_5 that we have in `reverse` between the first application of statement st_3: `y = x` and the first application of statement st_4: `x = x->n` in Figure 3. This results in three structures:

- The structure $S_{5,f,0}$, in which $\varphi_x^{st_4}(v)$ evaluates to 0 for all individuals. This structure represents a situation in which the concrete list that `x` and `y` point to has only one element, but the store also contains garbage cells, represented by summary node u.
- The structure $S_{5,f,1}$, in which $[\![\varphi_x^{st_4}(v)]\!]_3^{S_{5,f,1}}([v \mapsto u])$ equals 1. This covers the case where the list that `x` and `y` point to is a list of exactly two elements: In all of the concrete cells that summary node u represents, $\varphi_x^{st_4}(v)$ must evaluate to 1, and so u must represent just a single list node.
- The structure $S_{5,f,2}$, in which $[\![\varphi_x^{st_4}(v)]\!]_3^{S_{5,f,2}}([v \mapsto u.0])$ equals 0 and $[\![\varphi_x^{st_4}(v)]\!]_3^{S_{5,f,2}}([v \mapsto u.1])$ equals 1. This covers the case where the list that `x` and `y` point to is a list of three or more elements: In all of the concrete cells that $u.0$ represents, $\varphi_x^{st_4}(v)$ must evaluate to 0, and in all of the cells that $u.1$ represents, $\varphi_x^{st_4}(v)$ must evaluate to 1.

  This case captures the essence of node materialization as described in [19]: individual u is bifurcated into two individuals.

Notice how $focus_{\varphi_x^{st_4}(v)}$ is effectively constructed from S_5 by considering the reasons why $[\![\varphi_x^{st_4}(v)]\!]_3^{S_5}(Z)$ evaluates to 1/2 for a possible assignment Z: $[\![\varphi_x^{st_4}(v)]\!]_3^{S_5}([v \mapsto u_1])$ equals 0, and therefore $\varphi_x^{st_4}(v)$ is already in focus at u_1; in contrast, $[\![\varphi_x^{st_4}(v)]\!]_3^{S_5}([v \mapsto u])$ equals 1/2. There are three (maximal) structures in which $[\![\varphi_x^{st_4}(v)]\!]_3([v \mapsto u])$ has a definite value:

- $S_{5,f,0}$, in which $n(u_1, u)$ was forced to 0, and thus $[\![\varphi_x^{st_4}(v)]\!]_3^{S_{5,f,0}}([v \mapsto u])$ equals 0.
- $S_{5,f,1}$, in which $n(u_1, u)$ was forced to 1, and thus $[\![\varphi_x^{st_4}(v)]\!]_3^{S_{5,f,1}}([v \mapsto u])$ equals 1.
- $S_{5,f,2}$, in which u was bifurcated into two different individuals, $u.0$ and $u.1$. In $S_{5,f,2}$, $n(u_1, u.0)$ was set to 0, and thus $[\![\varphi_x^{st_4}(v)]\!]_3^{S_{5,f,2}}([v \mapsto u.0])$ equals 0, whereas $n(u_1, u.1)$ was set to 1, and thus $[\![\varphi_x^{st_4}(v)]\!]_3^{S_{5,f,2}}([v \mapsto u.1])$ equals 1.

Of course, there are other structures that can be embedded into S_5 that would assign a definite value to $\varphi_x^{st_4}(v)$, but these are not maximal (according to Definition 5.1) because each of them can be embedded into one of $S_{5,f,0}$, $S_{5,f,1}$, or $S_{5,f,2}$. □

In this paper, we simplify the analysis algorithm by only applying *focus* with respect to φ_x^{st} formulae, which ensures that the number of resulting structures is finite:

Lemma 5.4 *For every program variable* $\mathtt{x} \in PVar$, *statement* st, *and structure* S, $|focus_{\varphi_x^{st}}(S)| \leq 3|U^S|$. □

5.1.2 An Algorithm for Focus

In this section, we present an algorithm that implements $focus_{\varphi_x^{st}(v)}$ by generating structures in which $\varphi_x^{st}(v)$ has a definite value. A key aspect of the algorithm is the ability to identify the maximal structures in which $\varphi_x^{st}(v)$ has a definite value. Recall that the Hoare order on sets of structures is only a pre-partial order (see Definition 4.12). The following definition provides a way to compute a least upper bound and a greatest lower bound on sets of structures sharing the same universe U.

Definition 5.5 *Let* $XS_1, XS_2 \subseteq$ *3-STRUCT*$[\mathcal{P}]$ *such that for all* $S \in XS_1 \cup XS_2$, $U^S = U$. *We define the following operations on* XS_1 *and* XS_2:

$$XS_1 \sqcup XS_2 \stackrel{\text{def}}{=} maximal(XS_1 \cup XS_2)$$

$$XS_1 \sqcap XS_2 \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \langle U, \lambda p.\lambda u_1, \ldots u_k.\iota^{S_1}(p)(u_1, \ldots, u_k) \sqcap \iota^{S_2}(p)(u_1, \ldots, u_k)\rangle \\ \quad \mid S_1 \in XS_1 \textit{ and } S_2 \in XS_2 \textit{ and } comparable(S_1, S_2) \end{array} \right\}$$

where:

$$\begin{array}{l} comparable(S_1, S_2) = \\ \quad \textit{for all } p \in \mathcal{P}, \textit{for all } u_1, \ldots, u_k \in U: \\ \quad \iota^{S_1}(p)(u_1, \ldots, u_k) \sqsubseteq \iota^{S_2}(p)(u_1, \ldots, u_k) \\ \quad \textit{or } \iota^{S_2}(p)(u_1, \ldots, u_k) \sqsubseteq \iota^{S_1}(p)(u_1, \ldots, u_k) \end{array}$$

□

We are now ready to define the operations z and o that assure that a given formula evaluates to 0 and 1, respectively, in a given assignment.

Definition 5.6 *Let* $S = \langle U, \iota\rangle \in$ *3-STRUCT*$[\mathcal{P}]$ *be a three-valued structure. Let* $\iota[p(u_1, \ldots, u_k) \leftarrow l]$ *denote the map obtained from* ι *by updating* $\iota(p)(u_1, \ldots, u_k)$ *to have the value* l. *For a formula* φ *and assignment* Z, *we define* $z(\varphi)(S, Z) \in 2^{\text{3-STRUCT}[\mathcal{P}]}$ *and* $o(\varphi)(S, Z) \in 2^{\text{3-STRUCT}[\mathcal{P}]}$ *inductively, as follows:*

$$z(\boldsymbol{l})(S, Z) = \begin{cases} \{S\} & \text{if } \boldsymbol{l} = \mathbf{0} \\ \emptyset & \text{otherwise} \end{cases}$$

$$z(v_1 = v_2)(S, Z) = \begin{cases} \{S\} & \text{if } Z(v_1) \neq Z(v_2) \\ \emptyset & \text{otherwise} \end{cases}$$

$$z(p(v_1, \ldots, v_k))(S, Z) = \begin{cases} \{\langle U, \iota[p(Z(v_1), \ldots, Z(v_k)) \leftarrow 0]\rangle\} \\ \quad \text{if } 0 \sqsubseteq \iota(p(Z(v_1), \ldots, Z(v_k))) \\ \emptyset \quad \text{otherwise} \end{cases}$$

$$\begin{aligned} z(\varphi_1 \wedge \varphi_2)(S, Z) &= z(\varphi_1)(S, Z) \sqcup z(\varphi_2)(S, Z) \\ z(\varphi_1 \vee \varphi_2)(S, Z) &= z(\varphi_1)(S, Z) \sqcap z(\varphi_2)(S, Z) \\ z(\neg\varphi)(S, Z) &= o(\varphi)(S, Z) \\ z(\forall v : \varphi)(S, Z) &= \bigsqcup_{u \in U} z(\varphi)(S, Z[v \mapsto u]) \\ z(\exists v : \varphi)(S, Z) &= \bigsqcap_{u \in U} z(\varphi)(S, Z[v \mapsto u]) \end{aligned}$$

$$o(\boldsymbol{l})(S, Z) = \begin{cases} \{S\} & \text{if } \boldsymbol{l} = \mathbf{1} \\ \emptyset & \text{otherwise} \end{cases}$$

$$o(v_1 = v_2)(S, Z) = \begin{cases} z(sm(v_1))(S, Z) & \text{if } Z(v_1) = Z(v_2) \\ \emptyset & \text{otherwise} \end{cases}$$

$$o(p(v_1, \ldots, v_k))(S, Z) = \begin{cases} \{\langle U, \iota[p(Z(v_1), \ldots, Z(v_k)) \leftarrow 1]\rangle\} \\ \quad \text{if } 1 \sqsubseteq \iota(p(Z(v_1), \ldots, Z(v_k))) \\ \emptyset \quad \text{otherwise} \end{cases}$$

$$\begin{aligned} o(\varphi_1 \wedge \varphi_2)(S, Z) &= o(\varphi_1)(S, Z) \sqcap o(\varphi_2)(S, Z) \\ o(\varphi_1 \vee \varphi_2)(S, Z) &= o(\varphi_1)(S, Z) \sqcup o(\varphi_2)(S, Z) \\ o(\neg\varphi)(S, Z) &= z(\varphi)(S, Z) \\ o(\forall v : \varphi)(S, Z) &= \bigsqcap_{u \in U} o(\varphi)(S, Z[v \mapsto u]) \\ o(\exists v : \varphi)(S, Z) &= \bigsqcup_{u \in U} o(\varphi)(S, Z[v \mapsto u]) \end{aligned}$$

□

Example 5.7 For the formula $\varphi_x^{st_4}(v)$, structure S_5 from Fig-

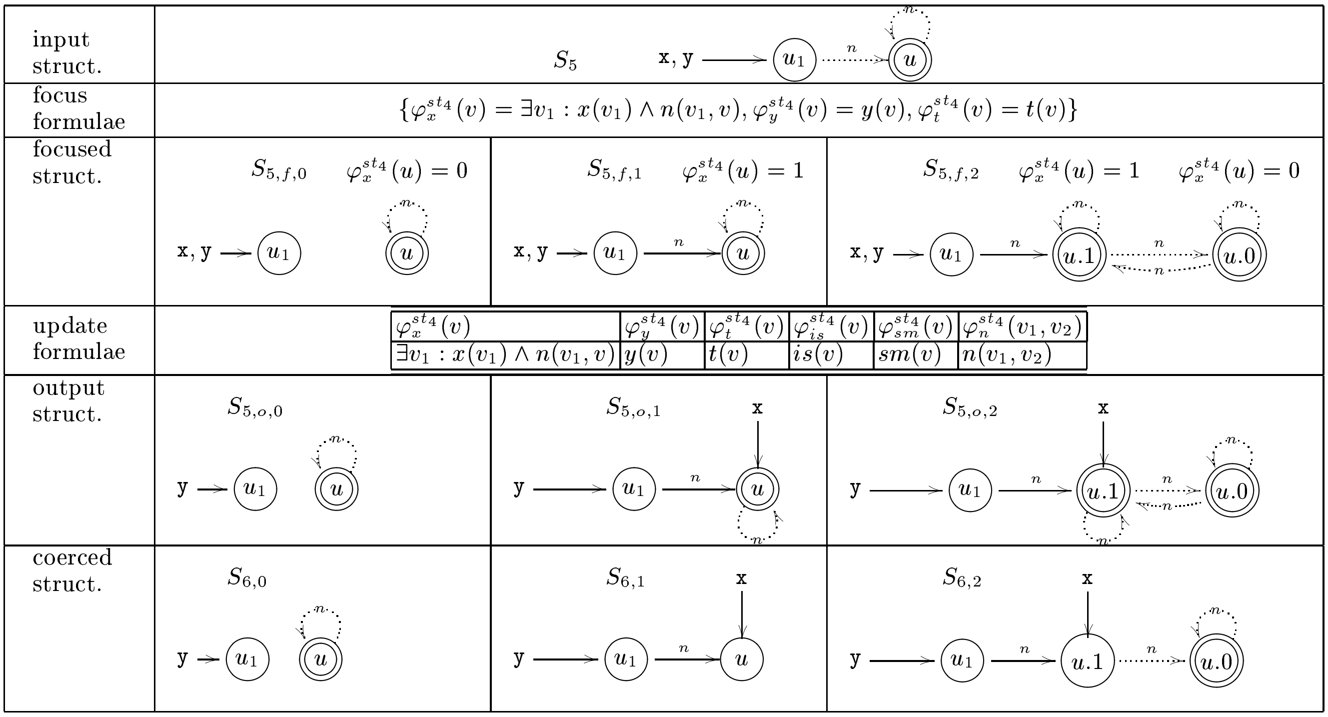


Figure 5: The first application of the improved transformer for statement st_4: `x = x->n` in `reverse`.

ure 5, and individual $u \in U^{S_5}$, we have:

$$
\begin{aligned}
& z(\varphi_x^{st_4})(S_5, [v \to u]) \\
&= z(\exists v_1 : x(v_1) \wedge n(v_1, v))(S_5, [v \to u]) \\
&= \bigsqcap_{u' \in \{u, u_1\}} z(x(v_1) \wedge n(v_1, v))(S_5, [v \to u, v_1 \to u']) \\
&= \begin{array}{l} z(x(v_1) \wedge n(v_1, v))(S_5, [v \to u, v_1 \to u]) \\ \sqcap\, z(x(v_1) \wedge n(v_1, v))(S_5, [v \to u, v_1 \to u_1]) \end{array} \\
&= \begin{array}{l} \left(\begin{array}{l} z(x(v_1)(S_5, [v \to u, v_1 \to u]) \\ \sqcup\, z(n(v_1, v))(S_5, [v \to u, v_1 \to u])) \end{array} \right) \\ \sqcap\, z(x(v_1) \wedge n(v_1, v))(S_5, [v \to u, v_1 \to u_1]) \end{array} \\
&= \begin{array}{l} (\{S_5\} \sqcup \langle \{u, u_1\}, \iota^{S_5}[n(u, u) \mapsto 0] \rangle) \\ \sqcap\, z(x(v_1) \wedge n(v_1, v))(S_5, [v \to u, v_1 \to u_1]) \end{array} \\
&= \{S_5\} \sqcap z(x(v_1) \wedge n(v_1, v))(S_5, [v \to u, v_1 \to u_1]) \\
&= \{S_5\} \sqcap \left(\begin{array}{l} (z(x(v_1))(S_5, [v \to u, v_1 \to u_1]) \\ \sqcup\, z(n(v_1, v))(S_5, [v \to u, v_1 \to u_1])) \end{array} \right) \\
&= \{S_5\} \sqcap z(n(v_1, v))(S_5, [v \to u, v_1 \to u_1]) \\
&= \{S_5\} \sqcap \langle \{u, u_1\}, \iota^{S_5}[n(u_1, u) \mapsto 0] \rangle \\
&= \{\langle \{u, u_1\}, \iota^{S_5}[n(u_1, u) \mapsto 0] \rangle\} \\
&= \{S_{5,f,0}\}
\end{aligned}
$$

Similarly, $o(\varphi_x^{st_4})(S_5, [v \to u]) = \{S_{5,f,1}\}$. $\square$

Remark. In Definition 5.6 we have ignored the case of formulae that include the transitive-closure operator. This was done both for notational simplicity, and because such formulae are not useful in the various predicate-update formulae φ_x^{st} employed by the abstract semantics. It is possible to handle such formulae by enumerating structures in which formulae evaluate to definite values. $\square$

The algorithm for *focus*, called Focus, is shown in Figure 6. When all of structure S's individuals have definite values for $\varphi_x^{st}(v)$, Focus returns $\{S\}$; when S has an individual u that has an indefinite value for $\varphi_x^{st}(v)$, Focus applies z and o to generate structures in which the indefiniteness is removed, and then recursively applies Focus to each of the structures generated. The call on auxiliary function Expand creates a structure in which individual u is bifurcated into two individuals; this captures the essence of shape-node materialization (cf. [19]).

function Focus(S : 3-STRUCT[$\mathcal{P}$], $\varphi_x^{st}(v)$: Formula)
returns $2^{\text{3-CSTRUCT}[\mathcal{P}, R(F)]}$
begin
  if there exists $u \in U^S$ s.t. $[\![\varphi_x^{st}]\!]_3^S([v \mapsto u]) = 1/2$ **then**
   let $u.0$ and $u.1$ be individuals not in U^S
   and $S' = o(\varphi_x^{st}(v))(z(\varphi_x^{st}(v))(\text{Expand}(S, u, u.0, u.1), [v \mapsto u.0]), [v \mapsto u.1])$
   and $XS = \begin{array}{l} z(\varphi_x^{st}(v))(S, [v \mapsto u]) \\ \cup\, o(\varphi_x^{st}(v))(S, [v \mapsto u]) \\ \cup\, S' \end{array}$
   in return $\bigcup_{S'' \in XS} \text{Focus}(S'', \varphi_x^{st}(v))$
  else return $\{S\}$
end

function Expand(S : 3-STRUCT[$\mathcal{P}$], $u, u.0, u.1$: elements)
returns 3-STRUCT[$\mathcal{P}$]
let $m = \lambda u'. \begin{cases} u & \text{if } u' = u.0 \vee u' = u.1 \\ u' & \text{otherwise} \end{cases}$ **in**
  return $\left\langle \begin{array}{l} (U^S - \{u\}) \cup \{u.0, u.1\} \\ \lambda p. \lambda u_1, \ldots, u_k. \iota^S(p)(m(u_1), \ldots, m(u_k)) \end{array} \right\rangle$

Figure 6: An algorithm for $focus_{\varphi_x^{st}(v)}$.

Example 5.8 Consider the application of Focus to the structure S_5 from Figure 5 and the formula $\varphi_x^{st_4}$. By Example 5.7, $z(\varphi_x^{st_4})(S_5, Z)$ yields the singleton set $\{S_{5,f,0}\}$ and $o(\varphi_x^{st_4})(S_5, Z)$ yields the singleton set $\{S_{5,f,1}\}$. By a similar derivation, $o(\varphi_x^{st_4}(v))(z(\varphi_x^{st_4}(v))(\text{Expand}(S, u, u.0, u.1), [v \mapsto u.0]), [v \mapsto u.1])$ yields the singleton set $\{S_{5,f,2}\}$. Thus, the result of Focus($S_5, \varphi_x^{st_4}$) is the set $\{S_{5,f,0}, S_{5,f,1}, S_{5,f,2}\}$. $\square$

5.2 Coercing into More Precise Structures

After *focus*, we apply the simple transformer $[\![st]\!]$ that was defined in Definitions 4.3 and 4.5. In the example discussed in Section 5.1, we apply $[\![st_4]\!]$ to the structures $S_{5,f,0}$, $S_{5,f,1}$, and $S_{5,f,2}$. We see that $S_{5,o,0}$ is obtained from $S_{5,f,0}$, $S_{5,o,1}$ from $S_{5,f,1}$, and $S_{5,o,2}$ from $S_{5,f,2}$.

Applying *focus* and then $[\![st]\!]$ can produce structures that are not as precise as we would like. The intuitive reason for this state of affairs is that there can be interdependences between different properties stored in a structure, and these interdependences are not necessarily incorporated in the definitions of the predicate-update formulae. This is demonstrated in the following example:

Example 5.9 Consider structure $S_{5,o,2}$ from Figure 5. In this structure, the `n` field of $u.0$ can point to $u.1$, which suggests that `x` may be pointing to a cyclic data structure. However, this is incompatible with the fact that $is(u.1) = 0$—i.e., $u.1$ cannot represent a heap-shared cell—and the fact that $n(u_1, u.1) = 1$—i.e., it is known that $u.1$ definitely has an incoming selector edge from a cell other than $u.0$. □

In this subsection, we show that in many cases we can sharpen the structures by removing indefinite values that violate certain compatibility rules. In particular, it allows us to remedy the imprecision illustrated in Example 5.9. Furthermore, the shape-analysis actually yields precise information in the analysis of `reverse`.

5.2.1 Compatibility Constraints

We can, in many cases, sharpen some of the stored predicate values of three-valued structures:

Example 5.10 Consider a two-valued structure $S^\natural$ that can be embedded in a three-valued structure S. By the Property-Extraction Principle (Observation 2.1), we know that if the formula φ_{is} for "inferring" whether an individual u is shared evaluates to, e.g., 1 in S, then in $S^\natural$, $is(u^\natural)$ must be 1 for any individual $u^\natural$ that maps to u. The definition of embedding (Definition 3.5) would allow the value of $is(u)$ in S to be 1/2; however, in this case a tighter embedding—in the sense of Definition 3.6—is also possible, in which $is(u)$ has the value 1. In other words, it is needlessly imprecise to let $is(u)$ retain the value 1/2: The "stored property" is should be at least as precise as its inferred value. Thus, in some cases, the fact that φ_{is} evaluates to 1 in a three-valued structure allows us to sharpen the stored predicate is.

Similar reasoning allows us to determine, in some cases, that a structure is inconsistent. For instance, if φ_{is} evaluates to 1 for an individual u and $is(u)$ is 0, then S is a three-valued structure that does not represent any concrete structures at all! When this situation arises, the structure can be eliminated from further consideration by the abstract-interpretation algorithm.

This reasoning applies to all instrumentation predicates, not just is, and to both of the definite values, 0 and 1. □

The reasoning used in Example 5.10 can be summarized as the following principle:

Observation 5.11 [The Sharpening Principle]. *In any structure S, the valued stored for $p(u_1, \ldots, u_k)$ should be at least as precise as the value of p's defining formula, φ_p, evaluated at $u_1, \ldots, u_k$ (i.e., $[\![\varphi_p]\!]^S_3([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k])$). Furthermore, if $p(u_1, \ldots, u_k)$ has a definite value and φ_p evaluates to an incomparable definite value, then S is a three-valued structure that does not represent any concrete structures at all.* □

$$\text{for each } \mathtt{x} \in PVar, x(v_1) \wedge x(v_2) \rhd v_1 = v_2 \quad (27)$$
$$(\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2)) \rhd v_1 = v_2 \quad (28)$$
$$(\exists v_1, v_2 : v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v)) \rhd is(v) \quad (29)$$
$$\neg(\exists v_1, v_2 : v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v)) \rhd \neg is(v) \quad (30)$$

$$(\exists v_1 : \neg is(v) \wedge v_1 \neq v_2 \wedge n(v_1, v)) \rhd \neg n(v_2, v) \quad (31)$$
$$(\exists v_2 : \neg is(v) \wedge v_1 \neq v_2 \wedge n(v_2, v)) \rhd \neg n(v_1, v) \quad (32)$$
$$(\exists v : \neg is(v) \wedge n(v_1, v) \wedge n(v_2, v)) \rhd v_1 = v_2 \quad (33)$$

Table 7: The compatibility constraints $R(\widehat{closure}(F))$ generated using Definition 5.13 from the formulae F given above the line in Table 4. The constraints below the line come from applying r to the formulae listed below the line in Table 4.

This observation motivates the subject of the remainder of this subsection—an investigation of compatibility constraints expressed in terms of a new logical connective, '$\rhd$'.

Definition 5.12 *Let Σ be a finite set of compatibility constraints of the form $\varphi_1 \rhd \varphi_2$, where φ_1 is an arbitrary three-valued formula, and φ_2 is either an atomic formula or a negation of an atomic formula. We say that a structure S* **satisfies** *Σ (denoted by $S \models \Sigma$) if for every constraint $\varphi_1 \rhd \varphi_2$ in Σ, and for every assignment Z such that $[\![\varphi_1]\!]^S_3(Z) = 1$, we have $[\![\varphi_2]\!]^S_3(Z) = 1$.* □

For a two-valued structure, $\rhd$ has the same meaning as $\Rightarrow$. However, for a three-valued structure $\rhd$ is stronger than $\Rightarrow$: if φ_1 evaluates to 1 and φ_2 evaluates to 1/2, the constraint $\varphi_1 \rhd \varphi_2$ is not satisfied. More precisely, suppose that $[\![\varphi_1]\!]^S_3(Z) = 1$ and $[\![\varphi_2]\!]^S_3(Z) = 1/2$; the implication $\varphi_1 \Rightarrow \varphi_2$ is satisfied (i.e., $S, Z \models \varphi_1 \Rightarrow \varphi_2$), but the constraint $\varphi_1 \rhd \varphi_2$ is not satisfied (i.e., $S, Z \not\models \varphi_1 \rhd \varphi_2$).

The constraint that captures the reasoning used in Example 5.10 is $\varphi_{is}(v) \rhd is(v)$. That is, when φ_{is} evaluates to 1 at u, then is must evaluate to 1 at u.

Such constraints formalize the Sharpening Principle. They will be used to improve the precision of the shape-analysis algorithm by (i) sharpening the values of stored predicates, and (ii) eliminating structures that violate the constraints.

The following definition converts formulae into constraints in a natural way:

Definition 5.13 *For formula φ and atomic formula a, define $r(\varphi)$ as follows.*

$$r(\forall v_1, \ldots v_k : (\varphi \Rightarrow a)) \stackrel{\text{def}}{=} \varphi \rhd a \quad (24)$$
$$r(\forall v_1, \ldots v_k : (\varphi \Rightarrow \neg a)) \stackrel{\text{def}}{=} \varphi \rhd \neg a \quad (25)$$
$$r(\forall v_1, \ldots v_k : \varphi) \stackrel{\text{def}}{=} \neg\varphi \rhd \mathbf{0} \quad (26)$$

For a set of formulae F, we define $R(F)$ to be the set of constraints obtained by applying r to each of the formulae in F. □

Rule (26) was added to enable an arbitrary formula to be converted to a constraint.

Example 5.14 The constraints generated for the formulae that appear above the line in Table 4 are listed above the line in Table 7. □

In [18], we define a closure operator $\widehat{closure}(F)$ that generates certain logical consequences of a set F of compatibility formulae. For instance, the three formulae below the line in Table 4 are generated by $\widehat{closure}(F)$, where F is the set of formulae given above the line in Table 4. The corresponding compatibility constraints that are obtained from $R(\widehat{closure}(F))$ are listed below the line in Table 7.

Example 5.15 As we will see in Section 5.2.3, compatibility constraints play a crucial role in the shape-analysis algorithm. Without them the algorithm would often be unable to determine that the data structure being manipulated by a list-manipulation program is actually a list. In particular, constraint (31) allows us to do a more accurate job of materialization: When $is(u)$ evaluates to 0 and one incoming n edge is 1, to satisfy constraint (31) a second incoming n edge cannot have the value 1/2—it must have the value 0, i.e., no such edge exists (cf. Examples 5.9 and 5.19). This allows edges to be removed (safely) that a more naive materialization process would retain (cf. structures $S_{5,o,2}$ and $S_{6,2}$ in Figure 5), and permits the improved shape-analysis algorithm to generate more precise structures for `reverse` than the ones generated by the simple shape-analysis algorithm described in Sections 2.3 and 4. □

Henceforth, we assume that $\widehat{closure}$ has been applied to all sets of hygiene conditions.

Definition 5.16 (Compatible Three-Valued Structures). *The* **set of compatible three-valued structures** *3-CSTRUCT*$[\mathcal{P}, R(F)] \sqsubseteq$ *3-STRUCT*$[\mathcal{P}]$ *is defined by* $S \in$ *3-CSTRUCT*$[\mathcal{P}, R(F)]$ *iff* $S \models R(F)$. □

5.2.2 The Coerce Operation

We are now ready to show how the *coerce* operation works.

Example 5.17 Consider structure $S_{5,o,2}$ from Figure 5 again. The structure $S_{5,o,2}$ violates constraint (31) under the assignment $[v \mapsto u.1, v_1 \mapsto u_1, v_2 \mapsto u.0]$. Because $\iota(is)(u.1) = 0$, $u_1 \neq u.0$, and $\iota(n)(u_1, u.1) = 1$, yet $\iota(n)(u.0, u.1) = 1/2$, constraint (31) is not satisfied: The left-hand side evaluates to 1, whereas the right-hand side evaluates to 1/2. □

This example motivates the following definition:

Definition 5.18 *The operation*

coerce: *3-STRUCT*$[\mathcal{P}] \to$ *3-CSTRUCT*$[\mathcal{P}, R(F)] \cup \{\bot\}$

is defined as follows: coerce$(S) \stackrel{\text{def}}{=}$ *the maximal* S' *such that* $S' \sqsubseteq S$, $U^{S'} = U^S$, *and* $S' \in$ *3-CSTRUCT*$[\mathcal{P}, R(F)]$, *or* $\bot$ *if no such* S' *exists.* □

Example 5.19 The application of *coerce* to the structures $S_{5,o,0}, S_{5,o,1}$, and $S_{5,o,2}$ is shown in the bottom block of Figure 5. It yields $S_{6,0}, S_{6,1}$, and $S_{6,2}$, respectively.

There are important differences between the structures $S_{6,0}, S_{6,1}$, and $S_{6,2}$ that result from the improved transformer for statement st_4 : `x = x->n`, and the structure S_6 that is the result of the simple version of the transformer (see the fourth entry of Figure 3): x points to a summary node in S_6, whereas in none of $S_{6,0}, S_{6,1}$, and $S_{6,2}$ does x point to a summary node. □

5.2.3 The Coerce Algorithm

In this subsection, we describe an algorithm, Coerce, that implements the operation *coerce* defined in Section 5.2. This algorithm actually finds a maximal solution to a system of constraints of the form defined in Definition 5.12. It is convenient to partition these constraints into the following types:

$$\varphi(v_1, v_2, \ldots, v_k) \triangleright \boldsymbol{b} \quad (34)$$

$$\varphi(v_1, v_2, \ldots, v_k) \triangleright (v_1 = v_2)^b \quad (35)$$

$$\varphi(v_1, v_2, \ldots, v_k) \triangleright p^b(v_1, v_2, \ldots, v_k) \quad (36)$$

where $\boldsymbol{b} \in \{\mathbf{0}, \mathbf{1}, \mathbf{1/2}\}$ and the superscript notation means the following: $\varphi^1 \equiv \varphi$ and $\varphi^0 \equiv \neg\varphi$. We say that constraints in the forms (34), (35), and (36) are *Type I*, *Type II*, and *Type III* constraints, respectively.

The algorithm for *coerce* is shown in Figure 7. The input is a three-valued structure $S \in$ 3-STRUCT$[\mathcal{P}]$ and a set of constraints $R(F)$. It initializes S' to the input structure S and then repeatedly refines S' by lowering predicate values in $\iota^{S'}$ from 1/2 into a definite value, until either: (i) a constraint is irreparably violated, i.e., the left-hand and the right-hand side have different definite values, in which case the algorithm returns $\bot$, or (ii) no constraint is violated, in which case the algorithm successfully returns S'. The main loop is a case switch on the type of the constraint considered:

- A violation of a Type I constraint is irreparable since the right-hand side is a literal.
- A violation of a Type II constraint can be fixed only when the right-hand side is an equality (as opposed to a negated equality) that evaluates to 1/2. This can happen when there is an individual u that is a summary node:

  $$[\![v_1 = v_2]\!]_3^{S'}([v_1 \mapsto u, v_2 \mapsto u]) = \iota^{S'}(sm)(u) = 1/2.$$

  In this case, $\iota^{S'}(sm)(u)$ is set to 0.
- A violation of a Type III constraint can be fixed when the predicate entry is indefinite.

The correctness of algorithm Coerce stems from the following lemma:

Lemma 5.20 *For every* $S, S_1 \in$ *3-STRUCT*$[\mathcal{P}]$, *such that* $S_1 \sqsubseteq S$ *and* $S_1 \models R(F)$, *and for every structure* S' *during each iteration of* Coerce, $S_1 \sqsubseteq S'$.

Proof: By induction on the number of iterations. □

Coerce must terminate after at most n steps, where n is the number of definite values in S', which is bounded by $\sum_{p \in \mathcal{P}} |U|^{arity(p)}$.

---

function Coerce(S: 3-STRUCT$[\mathcal{P}]$, $R(F)$: Constraint set)
returns 3-CSTRUCT$[\mathcal{P}, R(F)] \cup \{\bot\}$
begin
  $S' := S$
  while there exists a constraint $c \equiv \varphi_1 \triangleright \varphi_2 \in R(F)$ and an assignment Z: $freeVars(c) \to U^S$ such that $S', Z \not\models c$ **do**
    switch φ_2
      case $\varphi_2 \equiv \boldsymbol{b}$ /* Type I */
        return $\bot$
      case $\varphi_2 \equiv (v_1 = v_2)^b$ /* Type II */
        if $b = 1$ and $Z(v_1) = Z(v_2)$ and
          $\iota^{S'}(sm)(Z(v_1)) = 1/2$ **then** $\iota^{S'}(sm)(Z(v_1)) := 0$
        else return $\bot$
      case $\varphi_2 \equiv p^b(v_1, \ldots, v_k)$ /* Type III */
        if $\iota^{S'}(p)(Z(v_1), \ldots, Z(v_k)) = 1/2$ **then**
          $\iota^{S'}(p)(Z(v_1), \ldots, Z(v_k)) := b$
        else return $\bot$
    end switch
  od
  return S'
end

Figure 7: An iterative algorithm for solving three-valued constraints.

6 Related Work

The following previous shape-analysis algorithms, which all make use of some kind of shape-graph formalism, can be viewed as instances of our framework:

- The algorithm of [11], which collapses individuals that are not reachable from a pointer variable in k or fewer steps, for some fixed k. This algorithm can be captured in our framework by using instrumentation predicates of the form "reachable-from-x-via-access-path-α", for $|\alpha| \leq k$.
- The algorithms of [12, 2], which can be incorporated into the framework by introducing unary core predicates that record the allocation sites of heap cells.
- The algorithm of [16], which can be captured in the framework using the predicates $c_{f.b}(v)$ and $c_{b.f}(v)$ (see Tables 2 and 3).
- The algorithms of [22, 19]. These map unbounded-size stores into bounded-size abstractions by collapsing concrete cells that are not directly pointed to by program variables into one abstract cell, whereas concrete cells that are pointed to by different sets of variables are kept apart in different abstract cells. (See also the discussion in Section 4.3.)

Throughout this paper, we have focused on precision and ignored efficiency. The above-cited algorithms are more efficient than the one presented in this paper; however, Section 1.2 discusses reasons why it should be possible to incorporate well-known techniques for improving efficiency into our approach. In addition, the techniques presented in this paper may also provide a new basis for improving the efficiency of shape-analysis algorithms. In particular, the machinery we have introduced provides a way both to collapse individuals of 3-valued structures, via embedding, as well as to materialize them when necessary, via *focus*.

Roughly speaking, the chief alternative to the use of shape graphs involves representing may-aliases between pointer-access paths [8, 14, 4, 5, 20]. Compared with shape graphs, these methods have certain drawbacks. In particular, shape graphs represent the topological properties of the store directly, which allows certain operations, such as destructive updates, to be tracked more precisely. In addition, shape graphs are a more intuitive mechanism for reporting information back to a human, and thus may be more useful in program-understanding tools. On the other hand, representations of may-aliases can be more compact than shape graphs, and some may-alias algorithms are capable of representing information that goes beyond the capabilities of bounded structures [4, 5].

Acknowledgements

We are grateful for the helpful comments of A. Avron, T. Ball, M. Benedikt, N. Dor, M. Gitik, K. Kunen, V. Lifschitz, H.R. Nielson, M. O'Donnell, A. Rabinovich, and K. Sieber. We thank K.H. Rose for the Xy-pic LaTeXpackage.

References

[1] U. Assmann and M. Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models For Massively Parallel Computers*, pages 74–82, Washington, DC, September 1993. IEEE Press.

[2] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.

[3] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[4] A. Deutsch. A storeless model for aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE International Conference on Computer Languages*, pages 2–13, Washington, DC, 1992. IEEE Press.

[5] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.

[6] L. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.

[7] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 249–260, New York, NY, June 1992. ACM Press.

[8] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Par. and Dist. Syst.*, 1(1):35–47, January 1990.

[9] C.A.R. Hoare. Recursive data structures. *Int. J. of Comp. and Inf. Sci.*, 4(2):105–132, 1975.

[10] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 28–40, New York, NY, 1989. ACM Press.

[11] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[12] N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Symp. on Princ. of Prog. Lang.*, pages 66–74, New York, NY, 1982. ACM Press.

[13] S.C. Kleene. *Introduction to Metamathematics.* North-Holland, second edition, 1987.

[14] W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *Symp. on Princ. of Prog. Lang.*, pages 93–103, New York, NY, January 1991. ACM Press.

[15] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 21–34, New York, NY, 1988. ACM Press.

[16] J. Plevyak, A.A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lec. Notes in Comp. Sci.*, pages 37–57, Portland, OR, August 1993. Springer-Verlag.

[17] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Symp. on Princ. of Prog. Lang.*, New York, NY, January 1996. ACM Press.

[18] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. Tech. Rep. TR-1383, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, July 1998. Available at "http://www.cs.wisc.edu/wpis/papers/parametric.ps".

[19] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, January 1998.

[20] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to data flow analysis problems. *Acta Inf.*, 35(6):457–504, June 1998.

[21] J. Stransky. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Inf. and Comp.*, 101(1):70–102, Nov. 1992.

[22] E. Y.-B. Wang. *Analysis of Recursive Types in an Imperative Language.* PhD thesis, Univ. of Calif., Berkeley, CA, 1994.