Parametric Shape Analysis via 3-Valued Logic

Mooly Sagiv,[1] Thomas Reps,[2] and Reinhard Wilhelm[3]

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 holds in every store, in some store, or in no store 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. 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.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*; *dynamic storage management*; D.3.4 [**Programming Languages**]: Processors—*optimization*; E.1 [**Data**]: Data Structures—*graphs*; *lists*; *trees*; E.2 [**Data**]: Data Storage Representations—*composite structures*; *linked representations*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions*; *invariants*; *mechanical verification*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*mechanical theorem proving*

General Terms: Algorithms, Languages, Theory, Verification

Additional Key Words: Abstract interpretation, alias analysis, constraint solving, dataflow analysis, destructive updating, pointer analysis, shape analysis, static analysis, three-valued logic

---

[1]Part of this research was done while Sagiv was visiting the University of Chicago; part was done while visiting the Universität des Saaarlandes; part was done while visiting the University of Wisconsin, supported by the National Science Foundation under grant CCR-9619219. Sagiv is also supported by the United States-Israel Binational Science Foundation under grant 96-00337. Address: Dept. of Computer Science; School of Mathematical Sciences; Tel-Aviv University; Tel-Aviv 69978; Israel. Tel: +972-3-640-7606, Fax:+972-640-6761; E-mail: sagiv@math.tau.ac.il.

[2]Reps is supported in part by the National Science Foundation under grants CCR-9625667 and CCR-9619219, by the United States-Israel Binational Science Foundation under grant 96-00337, by grants from Rockwell and IBM, and by a Vilas Associate Award from the University of Wisconsin. Address: Computer Sciences Department; University of Wisconsin; 1210 West Dayton Street; Madison, WI 53706; USA. Tel: +1-608-262-2091, Fax:+1-608-262-9777; E-mail: reps@cs.wisc.edu.

[3]Wilhelm is supported in part by a DAAD-NSF Collaborative Research Grant. Address: Fachbereich 14 Informatik; Universität des Saarlandes; 66123 Saarbrücken; Germany. Tel: +49-681-302-4399, Fax: +49-681-302-3065; E-mail: wilhelm@cs.uni-sb.de.

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 [Hoa75]. Such invariants are useful for sharpening the results obtained from a tool like `LClint`, which predicts memory-usage bugs [Eva96], and for program optimization (e.g., to improve memory locality [LM96]).

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 [JM81, JM82, LH88, HPR89, CWZ90, Str92, AW93, PCK93, Wan94, SRW98]. A common feature of these algorithms is that they represent multiple run-time locations by a single "shape-node", often called *summary-nodes* [CWZ90]. 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. Such a framework has two parts: (i) a language for specifying different abstraction properties and how they are affected by the execution of the different kinds of statements in the programming language, and (ii) a method for generating a shape-analysis algorithm from such a description. The first is an issue having to do with *specification*; the specified set of properties determines the set of observable data structures. The second is an issue of how to generate an appropriate algorithm from the specification. 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 many programs, including ones that manipulate several (possibly cyclic) data structures simultaneously (see Section 6). Other static-analysis techniques (including ones that are not based on shape graphs [LR91, Hen90, HN90, Deu92, Deu94]) yield very imprecise information on these programs.

1.1.1 The Use of Logic for Shape Analysis

In the shape-analysis framework, predicate-logic formulae play many roles: expressing both the concrete and abstract semantics of the programming language, specifying the parameters used for instantiating the parametric framework in different ways, expressing properties of store elements (e.g., may-aliases, must-aliases), and expressing properties of stores (e.g., data-structure invariants). For example, to specify the operational semantics of the statement '`x = x->n`', where `x` is a pointer variable and `n` is a pointer field, 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 run-time locations that formerly were 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 [Kle87] (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 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 [JM81, HPR89, CWZ90, Str92].

1.1.3 Materialization of New Nodes from Summary Nodes

One of the magical aspects of [SRW98] is "materialization", in which a transfer function splits a summary-node into two separate nodes. (This operation is also discussed in [CWZ90, PCK93].) 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 [SRW98], it was observed that node materialization is complicated because various kinds of shape-graph properties are linked. For instance, the heap-sharing properties of shape graphs 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 property linkages. In Section 5.2.3, we give an algorithm that solves systems of such constraints.

1.2 Limitations

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

- There are other algorithms—not based on shape graphs—that are incomparable to our method. For example, the algorithm of [Hen90, HN90] handles destructive updates in many cases, but does not handle cyclic or doubly-linked lists. The algorithm of Deutsch [Deu92, Deu94] is able to represent cyclic lists but fails to handle many kinds of destructive-update operations (due to the absence of must-alias information).

- The framework creates intraprocedural shape-analysis algorithms, not interprocedural ones. Methods for handling procedures are presented in [CWZ90, AW93, SRW98]. 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.

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

(a)

```c
/* 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;
}
```

(b)

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`.

- The number of shape graphs may be quite large (as in [JM81, HPR89]). This problem was avoided in [LH88, CWZ90, PCK93, SRW98] 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.

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. Section 6 discusses two particularly important instantiations of the parametric framework, both of which involve tracking reachability properties. Two proofs are presented in Appendices A and B.

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

The standard logical interpretation used in many of the classical dataflow-analysis algorithms, as well as in shape analysis [SRW98], uses two values. It is definite on one of the values and conservative on the other. That is, either "false" means "false" and "true" means "may be true/may be false, or "true" means "true" and "false" means "may be false/may be true". With three-valued logic, definite truth and definite falseness can both be tracked, with the third value, "unknown", representing uncertainty, i.e., the fact that the analysis is unable to give a definite answer.

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 $\langle U, \iota \rangle$.

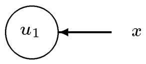
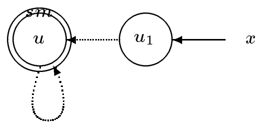
<table>
<tr><th>S</th><th>Structure</th><th>Graphical Representation</th></tr>
<tr><td>S_0</td><td>
unary
<table><tr><th>el.</th><th>x</th><th>y</th><th>t</th><th>sm</th><th>is</th></tr></table>
n
</td><td></td></tr>
<tr><td>S_1</td><td>
unary
<table><tr><th>el.</th><th>x</th><th>y</th><th>t</th><th>sm</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>
n
<table><tr><th></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
<table><tr><th>el.</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>
n
<table><tr><th></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`.

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 $p(u_1, u_2, \ldots, u_k)$ 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 predicates.

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. Unary predicates with value 1 are layered around the circle with solid circles, unary predicates with value 1/2 with dotted circles ("onion-skin" notation). Unary pointed-to-by-`x` predicates with value 1 (1/2) and binary pointer-component-points-to predicates with value 1 (1/2) 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.[4]

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. Thus, $x(u_1) = 1$, because `x` points to u_1, which represents the head of the list. Also, $y(u) = y(u_1) = t(u) = t(u_1) = 0$ because `y` and `t` do not point to any cell of heap-allocated storage.
- 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.

[4]Due to typographical difficulties at the time of writing, dotted circles appear as solid circles. In the graphical representations of all example structures that appear in the paper, every circle labeled by sm or is should be shown as a dotted circle.

$\wedge$	0	1	1/2
0	0	0	0
1	0	1	1/2
1/2	0	1/2	1/2

$\vee$	0	1	1/2
0	0	1	1/2
1	1	1	1
1/2	1/2	1	1/2

$\neg$	
0	1
1	0
1/2	1/2

Table 1: Kleene's three-valued interpretation of the propositional operators.

- 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.

The three-valued structures deliberately ignore the following properties of concrete lists:

- The actual values of fields of data-structure cells, e.g., the values in the `data` fields.
- The actual length of lists. For example, S_2 represents all the lists with two or more elements.

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.*

□

Kleene's three-valued interpretation of the propositional operators is given in Table 1.

In Section 3.4, we give the *Embedding Theorem* (Theorem 3.11), 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. This provides the basis for using the results of shape analysis in optimization. For example, for all abstract elements of structure S_2, the formula

$$\exists v : x(v) \wedge n(v, v),$$

which expresses the property "`x` points to a cell that has a self-cycle", evaluates to 0 because `x` and `x->n` point to different elements in all of the stores represented by S_2. This information can be used by an optimizing compiler to determine whether it is profitable to generate a prefetch for the next element [LM96].

Now consider the formula

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

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 $1/2$ in Kleene's semantics. 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.* □

As we will see shortly, the use of instrumentation predicates also allows us to define a *parametric framework* for shape analysis. In general, adding additional instrumentation predicates refines the abstraction: It yields a more precise shape-analysis algorithm that maintains finer distinctions, and hence allows more questions about the program's heap-allocated data structures to be answered.

2.3 Simple Abstract Interpretation of Program Statements

The most complex issue that we face is the definition of the abstract semantics of program statements. This abstract semantics has to be (i) conservative, i.e., must represent every possible run-time situation, and (ii) should not yield too many "unknown" values.

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).* □

Evaluation of the formulae in two-valued logic captures the concrete semantics. Evaluation of the formulae in three-valued logic captures the abstract semantics.

Observation 2.3 allows us to simplify drastically the argument that the shape-analysis framework is correct, because the correctness of the abstract semantics falls out directly from the Embedding Theorem.

Table 2 illustrates the steps 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 Table 2. Table 2 lists a predicate-update formula $p'(v)$ only if predicate p is affected by the execution of the statement. For any unchanged predicate q, the predicate-update formula is "$q'(v) = q(v)$". For instance, statement st_1 sets `y` to `NULL`. The complete list of predicate-update formulae for st_1 is: $x'(v) = x(v)$, $y'(v) = \mathbf{0}$, $t'(v) = t(v)$, $n'(v_1, v_2) = n(v_1, v_2)$, $sm'(v) = sm(v)$, and $is'(v) = is(v)$. Thus, after st_1 program-variable `y` does not point to any element.

Table 2: The simple abstract interpretation of `reverse` applied to structure S_2 shown in Figure 2 (which represents lists of length two or more).

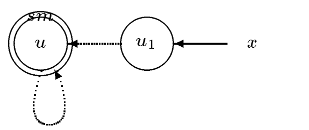
statement	formula	structure
st_1: `y = NULL;`	$y'(v) = \mathbf{0}$	S_3 

Table 2: The simple abstract interpretation of `reverse` applied to structure S_2 shown in Figure 2 (which represents lists of length two or more).

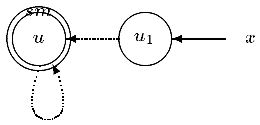
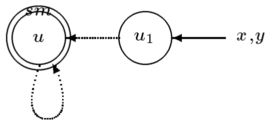
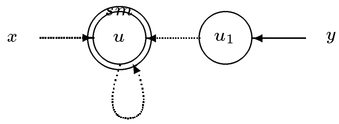
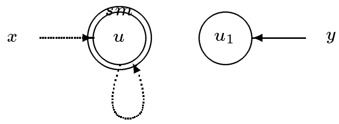
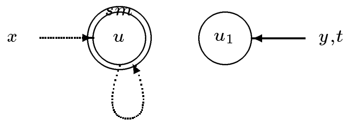
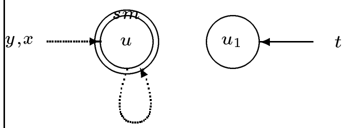
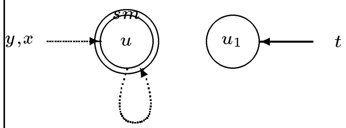
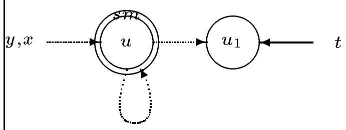
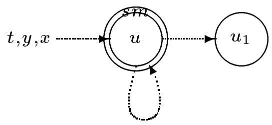
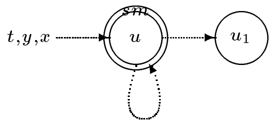
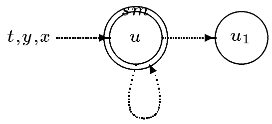
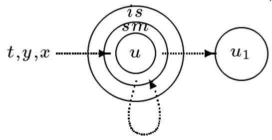
statement	formula	structure
st_2: `t = y;`	$t'(v) = y(v)$	S_4 
st_3: `y = x;`	$y'(v) = x(v)$	S_5 
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$	S_6 
st_5: `y->n = t;`	$n'(v_1, v_2) = (n(v_1, v_2) \wedge \neg y(v_1)) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = \exists v_1, v_2 : (is(v) \wedge n'(v_1, v) \wedge n'(v_2, v) \wedge v_1 \neq v_2) \vee (t(v) \wedge n(v_1, v) \wedge \neg y(v_1))$	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;`	$n'(v_1, v_2) = (n(v_1, v_2) \wedge \neg y(v_1)) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = \exists v_1, v_2 : (is(v) \wedge n'(v_1, v) \wedge n'(v_2, v) \wedge v_1 \neq v_2) \vee (t(v) \wedge n(v_1, v) \wedge \neg y(v_1))$	S_{11} 

Table 2: The simple abstract interpretation of `reverse` applied to structure S_2 shown in Figure 2 (which represents lists of length two or more).

statement	formula	structure
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;`	$n'(v_1, v_2) = (n(v_1, v_2) \wedge \neg y(v_1)) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = \exists v_1, v_2 : (is(v) \wedge n'(v_1, v) \wedge n'(v_2, v) \wedge v_1 \neq v_2) \vee (t(v) \wedge n(v_1, v) \wedge \neg y(v_1))$	S_{15} 

As we will see, this approach has a number of good properties:

- The abstract-interpretation process will always terminate if we guarantee that the number of elements in three-valued structures is bounded.
- The Embedding Theorem implies that the results obtained are conservative.
- By defining appropriate instrumentation predicates, it is possible to emulate some previous shape-analysis algorithms. The shape-analysis algorithm illustrated in Table 2 is essentially that of Chase et al. [CWZ90]. Others that are amenable to being simulated in this fashion include [JM81, LH88, HPR89].

Unfortunately, there is also bad news: The method described above and illustrated in Table 2 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_4, which produces structure S_{15}. Structure S_{15} indicates that "`x`, `y`, and `t` may all point to the same (possibly shared) list".

This provides insight into where the algorithm of Chase et al. loses precision.

2.4 Improved Abstract Interpretation of Program Statements

In Section 5, we show how it is possible to go beyond the simplistic approach described above in Section 2.3 by "materializing" new non-summary nodes from summary nodes as data structures are

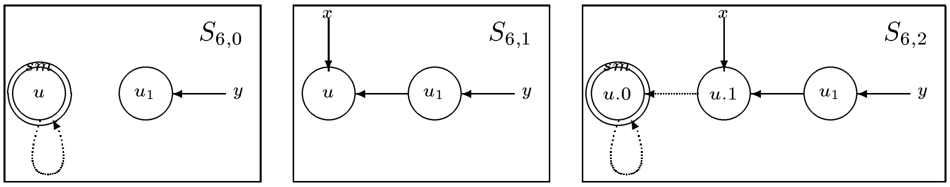


Figure 3: The three structures that result from the first abstract execution of st_4 by the improved abstract-interpretation method of Section 5.

traversed. (Thus, Section 5 generalizes the algorithm of [SRW96].) As we will see in Section 5, this allows us to determine the correct shape descriptors for the data structures used in the `reverse` program.

To perform a more precise abstract interpretation of programs, we have to be able to materialize new nodes from summary nodes as the program's data structures are traversed. Plevyak et al. [PCK93] introduced a way to do materialization for straight-line code, and Sagiv et al. [SRW98] developed a way to do this in the presence of loops and recursion. However, these analyses are hard to understand and to show correct.

In Section 5, we show a systematic solution to the materialization problem that is (i) relatively easy to understand and prove correct, and (ii) applies to an arbitrary set of predicates. It is based on the following principle:

Observation 2.4 [**Materialization Principle**]. *Materialization is driven by a mechanism that refines a three-valued structure into possibly several more-precise structures by forcing certain predicate values to have definite values, i.e.,* 0 *or* 1*. The abstract semantics described in Section 2.3 is then applied to the more-precise structures.* □

For instance, Figure 3 shows the three structures that result from the first abstract execution of st_4 by the improved abstract-interpretation method of Section 5. In contrast to the structure S_6 produced by the method of Section 2.3, for all elements in all of the structures that occur in Figure 3, $x(v)$ evaluates to 0 or 1 and not 1/2.

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 [**Information Order**]. *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 $l_2 = 1/2$.* □

Kleene's semantics of three-valued logic is monotonic in the information order (see Table 1 and Definition 3.4).

The values 0, 1, and 1/2 form a mathematical structure known as a bi-lattice, e.g., [Gin88], as shown in Figure 4. A bi-lattice has two orderings: the *logical order* and the *information order*. The logical order is the one used in Table 1: that is, $\wedge$ and $\vee$ are meet and join in the logical order (e.g., $1 \wedge 1/2 = 1/2$, $1 \vee 1/2 = 1$, $1/2 \wedge 0 = 0$, $1/2 \vee 0 = 1/2$, etc.). The information order is the one defined in Definition 3.1 to capture "(un)certainty".

In Figure 4, a value that is "far enough to the right" in the logical order indicates "potential truth" (and is called a *designated value*). In the bi-lattice of Figure 4 we take 1/2 and 1 as the designated values. This means that a structure potentially satisfies a formula when the formula's interpretation is either 1/2 or 1 (see Definition 3.4).

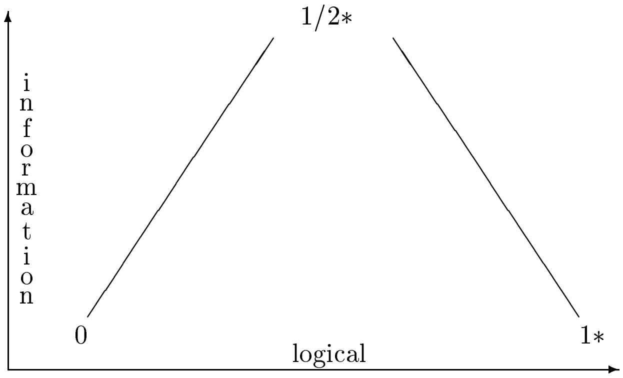


Figure 4: The bi-lattice of three-valued logic. (The $*$ symbols attached to 1/2 and 1 indicate that these are the "designated values", which indicate "potential truth".)

3.1 Syntax of First-Order Formulae with Transitive Closure

Let $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ be a finite set of predicate symbols. Without loss of generality we exclude constant and function symbols from our logic.[5] 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 $(TC\ v_1, v_2 : p(v_1, v_2))(v_3, v_4)$; $\varphi_1 \implies \varphi_2$ is a shorthand for $(\neg\varphi_1 \vee \varphi_2)$; and $\varphi_1 \iff \varphi_2$ is a shorthand for $(\varphi_1 \implies \varphi_2) \wedge (\varphi_2 \implies \varphi_1)$.

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

Definition 3.2 *A* **formula** *over a* **vocabulary** $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ *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, $p(v_1, v_2, \ldots, v_k)$ is an atomic formula with free variables $v_1, v_2, \ldots, v_k$.

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.* $\Box$

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 values of the core-predicates are dictated by the programming-language semantics. In contrast, instrumentation predicates are used in order to improve the precision of the analysis (as described by Observation 2.2).

Example 3.3 Table 3 contains the core-predicates for the `List` data-type declaration from Figure 1(a) and the `reverse` program of Figure 1(b). The unary predicate $sm \in \mathcal{C}$ captures the essence

[5] Constant symbols can be encoded via unary predicates and n-ary functions via $n+1$-ary predicates.

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

Pred.	Intended Meaning	Purpose	Ref.
$is(v)$	Do two or more fields of heap elements point to v?	lists and trees	[CWZ90, SRW98]
$r(v)$	Is v reachable from some pointer variable (i.e., is v a non-garbage element)?	compile-time garbage collection	
$r_x(v)$	Is v (transitively) reachable from pointer variable `x`?	separating disjoint data structures	[SRW98, p.38]
$c(v)$	Is v on a directed cycle?	reference counting	[JM81]
$c_{f.b}(v)$	Does a field-`f` dereference from v, followed by a field-`b` deference, yield v?	doubly-linked lists	[HHN92, PCK93]
$c_{b.f}(v)$	Does a field-`b` dereference from v, followed by a field-`f` deference, yield v?	doubly-linked lists	[HHN92, PCK93]

Table 4: Examples of instrumentation predicates.

of "summary-nodes", which were introduced by Jones and Muchnick [JM81] to represent an unbounded number of concrete elements by a single abstract element. There are two possible values for $sm(u)$:

- 0, when u represents a unique element. This is the case for all elements of concrete stores (because cells in a concrete store represent only themselves). It is also the case for abstract elements that are definitely pointed to by a pointer variable (because a pointer variable can only point to a single concrete element). For example, in structure S_2 from Figure 2, u_1 represents a unique concrete element of any store the S_2 represents—the element pointed to by variable `x`.

- 1/2, when u may or may not represent more than one element. For example, element u of structure S_2 represents a single concrete element if `x` points to a two-element list, but represents two or more concrete elements if `x` points to a list of length three or more.

- Intuitively, $sm(u) = 1$ should mean that u definitely represents more than one element. This is disallowed for technical reasons. In particular, allowing $sm(u)$ to be 1 violates the Property-Extraction Principle (Observation 2.1); this will become clearer in Sections 3.4 and 3.5.

It is instructive to consider a variant of structure S_2 from Figure 2: Let structure S_2' be identical to S_2 except that $sm(u) = 0$. S_2' represents lists of exactly two elements (but not lists of length three or more). □

Table 4 lists some interesting instrumentation predicates, and Table 5 lists their defining formulae.

- The sharing predicate is was introduced in [CWZ90] and also used in [SRW98] to capture list and tree data structures.

- The reachability predicate r identifies non-garbage cells. This is useful for determining when compile-time garbage collection can be performed.

- The reachability-from-`x` predicate r_x was mentioned in [SRW98, p.38]. It drastically improves the precision of shape analysis, even for programs that manipulate simple list and tree data

$$
\begin{aligned}
\varphi_{is}(v) &\stackrel{\text{def}}{=} \exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2 && (4)\\
\varphi_{r}(v) &\stackrel{\text{def}}{=} \bigvee_{\mathtt{x} \in PVar} (x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v)) && (5)\\
\varphi_{r_x}(v) &\stackrel{\text{def}}{=} x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v) && (6)\\
\varphi_{c}(v) &\stackrel{\text{def}}{=} n^+(v, v) && (7)\\
\varphi_{c_{f.b}}(v) &\stackrel{\text{def}}{=} \forall v_1, v_2 : f(v, v_1) \wedge b(v_1, v_2) \implies v_2 = v && (8)\\
\varphi_{c_{b.f}}(v) &\stackrel{\text{def}}{=} \forall v_1, v_2 : b(v, v_1) \wedge f(v_1, v_2) \implies v_2 = v && (9)
\end{aligned}
$$

Table 5: Formulae that define the meaning of the instrumentation predicates listed in Table 4.

structures, since it keeps separate the abstract representations of data structures that are disjoint in the concrete world.

- The cyclicity predicate c was introduced by Jones and Muchnick [JM81] 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 [HHN92] and also used in [PCK93].

In the general case, a program uses a number of different struct types. The core vocabulary is then defined as follows:

$$
\mathcal{C} \stackrel{\text{def}}{=} \{sel \mid \mathtt{sel} \in Sel\} \cup \{x \mid \mathtt{x} \in PVar\} \cup \{sm\}, \tag{10}
$$

where Sel is the set of pointer-valued fields in the struct types declared in the program, and $PVar$ is the set of pointer variables in the program. The formula for is is then

$$
\varphi_{is}(v) \stackrel{\text{def}}{=} \begin{array}{ll} & \displaystyle\bigvee_{\mathtt{sel} \in Sel} \exists v_1, v_2 : sel(v_1, v) \wedge sel(v_2, v) \wedge v_1 \neq v_2 \\ \vee & \displaystyle\bigvee_{\substack{\mathtt{sel}_1, \mathtt{sel}_2 \in Sel,\\ \mathtt{sel}_1 \neq \mathtt{sel}_2}} \exists v_1, v_2 : sel_1(v_1, v) \wedge sel_2(v_2, v). \end{array}
$$

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:*

$$
\iota^S \colon \mathcal{P} \to (U^S)^k \to \{0, 1, 1/2\}.
$$

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 $p(v_1, v_2, \ldots, v_k)$,

$$[\![p(v_1, v_2, \ldots, v_k)]\!]_3^S(Z) = \iota^S(p)(Z(v_1), Z(v_2), \ldots, Z(v_k))$$

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

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

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

$$\begin{aligned} [\![\varphi_1 \wedge \varphi_2]\!]_3^S(Z) &= \min([\![\varphi_1]\!]_3^S(Z), [\![\varphi_2]\!]_3^S(Z)) \\ [\![\varphi_1 \vee \varphi_2]\!]_3^S(Z) &= \max([\![\varphi_1]\!]_3^S(Z), [\![\varphi_2]\!]_3^S(Z)) \\ [\![\neg\varphi_1]\!]_3^S(Z) &= 1 - [\![\varphi_1]\!]_3^S(Z) \end{aligned}$$

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

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

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

$$[\![(TC\ v_1, v_2 : \varphi)(v_3, v_4)]\!]_3^S(Z) = \max_{\substack{u_1, u_2, \ldots, u_n \in U \\ Z(v_3) = u_1, Z(v_4) = u_n}} \min_{i=1}^{n-1} [\![\varphi]\!]_3^S(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

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*:* $S, Z \models \varphi$*.* □

Example 3.5 Consider the structure S_2 from Figure 2 and formula (4),

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

which expresses the property "Do two or more different cells point to v?". For the assignment $Z = [v \mapsto u]$,

$$\begin{aligned} [\![\varphi_{is}]\!]_3^S(Z) &= \max_{u', u'' \in \{u_1, u\}} [\![n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2]\!]_3^S([v \mapsto u, v_1 \mapsto u', v_2 \mapsto u'']) \\ &= 1/2, \end{aligned}$$

and thus $S_2, Z \models \varphi_{is}$. For the assignment $Z = [v \mapsto u_1]$,

$$\begin{aligned} [\![\varphi_{is}]\!]_3^S(Z) &= \max_{u', u'' \in \{u_1, u\}} [\![n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2]\!]_3^S([v \mapsto u_1, v_1 \mapsto u', v_2 \mapsto u'']) \\ &= 0, \end{aligned}$$

and thus $S_2, Z \not\models \varphi_{is}$. □

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 '$=$'):[6]

- 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 $1/2$.

[6]Note that there is only a small typographical distinction between the syntactic symbol for equality, namely '$=$', and the symbol for the "identically-equal" relation on individuals, namely '$=$'. Throughout the paper, it should always be clear from the context which symbol is intended.

3.3 Properties of Three-Valued Logic

Three-valued logic retains a number of properties that are familiar from two-valued logic:

Lemma 3.6 *Let φ_1, φ_2, and φ_3 be formulae, let S be a three-valued structure, and let Z be a complete assignment for the formula or formulae of interest. Then the following properties hold:*

Double-Negation

$$[\![\neg(\neg\varphi_1)]\!]_3^S(Z) = [\![\varphi_1]\!]_3^S(Z) \tag{11}$$

De Morgan Laws

$$\begin{aligned}
[\![\neg(\varphi_1 \wedge \varphi_2)]\!]_3^S(Z) &= [\![\neg\varphi_1 \vee \neg\varphi_2]\!]_3^S(Z) && (12)\\
[\![\neg(\varphi_1 \vee \varphi_2)]\!]_3^S(Z) &= [\![\neg\varphi_1 \wedge \neg\varphi_2]\!]_3^S(Z) && (13)\\
[\![\neg(\exists v : \varphi_1)]\!]_3^S(Z) &= [\![\forall v : \neg\varphi_1]\!]_3^S(Z) && (14)\\
[\![\neg(\forall v : \varphi_1)]\!]_3^S(Z) &= [\![\exists v : \neg\varphi_1]\!]_3^S(Z) && (15)
\end{aligned}$$

Associativity Laws

$$\begin{aligned}
[\![(\varphi_1 \wedge \varphi_2) \wedge \varphi_3]\!]_3^S(Z) &= [\![\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)]\!]_3^S(Z) && (16)\\
[\![(\varphi_1 \vee \varphi_2) \vee \varphi_3]\!]_3^S(Z) &= [\![\varphi_1 \vee (\varphi_2 \vee \varphi_3)]\!]_3^S(Z) && (17)
\end{aligned}$$

Commutativity Laws

$$\begin{aligned}
[\![\varphi_1 \wedge \varphi_2]\!]_3^S(Z) &= [\![\varphi_2 \wedge \varphi_1]\!]_3^S(Z) && (18)\\
[\![\varphi_1 \vee \varphi_2]\!]_3^S(Z) &= [\![\varphi_2 \vee \varphi_1]\!]_3^S(Z) && (19)
\end{aligned}$$

Distributivity Laws

$$\begin{aligned}
[\![\varphi_1 \wedge (\varphi_2 \vee \varphi_3)]\!]_3^S(Z) &= [\![(\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)]\!]_3^S(Z) && (20)\\
[\![\varphi_1 \vee (\varphi_2 \wedge \varphi_3)]\!]_3^S(Z) &= [\![(\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)]\!]_3^S(Z) && (21)
\end{aligned}$$

Implication Law

$$[\![\varphi_1 \implies \varphi_2]\!]_3^S(Z) = [\![\neg\varphi_2 \implies \neg\varphi_1)]\!]_3^S(Z) \tag{22}$$

□

Kleene's semantics is monotonic in the information order:

Lemma 3.7 *Let φ be a formula, and let S and S' be two structures such that $U^S = U^{S'}$ and $\iota^S \sqsubseteq \iota^{S'}$. (That is, for each predicate symbol p of arity k, $\iota^S(p)(u_1, u_2, \ldots, u_k) \sqsubseteq \iota^{S'}(p)(u_1, u_2, \ldots, u_k)$.) Then, for every complete assignment Z,*

$$[\![\varphi]\!]_3^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S'}(Z). \tag{23}$$

□

3.4 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 3.8 *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 $u_1, u_2, \ldots, u_k \in U^S$,*

$$\iota^S(p)(u_1, u_2, \ldots, u_k) \sqsubseteq \iota^{S'}(p)(f(u_1), f(u_2), \ldots, f(u_k)) \tag{24}$$

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

$$(|\{u \mid f(u) = u'\}| > 1) \sqsubseteq \iota^{S'}(sm)(u') \tag{25}$$

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 (24) 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 3.9 *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,*

$$\iota^{S'}(p)(u'_1, u'_2, \ldots, u'_k) = \bigsqcup_{\substack{t\_embed(u_1)=u'_1\\ t\_embed(u_2)=u'_2\\ \vdots\\ t\_embed(u_k)=u'_k}} \iota^S(p)(u_1, u_2, \ldots, u_k) \tag{26}$$

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

$$\iota^{S'}(sm)(u') = (|\{u | t\_embed(u) = u'\}| > 1) \sqcup \bigsqcup_{t\_embed(u)=u'} \iota^S(sm)(u) \tag{27}$$

Because t_embed is surjective, equations (26) and (27) uniquely determine S' (up to isomorphism); therefore, we say that $S' = t\_embed(S)$. □

It is immediately apparent from Definition 3.9 that the tight embedding of a structure S by a function t_embed possessing properties (26) and (27) embeds S in $t\_embed(S)$, i.e., $S \sqsubseteq^{t\_embed} t\_embed(S)$.

It is also apparent from Definition 3.9 how several individuals from U^S can "lose their identity" by being mapped to the same individual in $U^{S'}$:

Example 3.10 Let $u_1, u_2 \in U^S$, where $u_1 \neq u_2$, be individuals such that $\iota^S(sm)(u_1) = 0$ and $\iota^S(sm)(u_2) = 0$ both hold, and where $t\_embed(u_1) = t\_embed(u_2) = u'$. Therefore, $\iota^{S'}(sm)(u') = 1/2$, and consequently, $[\![v_1 = v_2]\!]_3^{S'}([v_1 \mapsto u', v_2 \mapsto u']) = 1/2$. In other words, we do not know if u' is equal to itself! □

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

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.

Formally, we have the following theorem:

Theorem 3.11 [**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,*

$$[\![\varphi]\!]_3^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S'}(f\, o\, Z)$$

Proof: Appears in Appendix A. □

Example 3.12 Continuing Example 3.10, we can illustrate the Embedding Theorem on the formula $\varphi \equiv v_1 = v_2$ and the embedding $f \equiv t\_embed$, as follows:

$$
\begin{array}{rcl}
0 & = & [\![v_1 = v_2]\!]_3^S([v_1 \mapsto u_1, v_2 \mapsto u_2]) \\
 & \sqsubseteq & [\![v_1 = v_2]\!]_3^{S'}(t\_embed\; o\, [v_1 \mapsto u_1, v_2 \mapsto u_2]) \\
 & = & [\![v_1 = v_2]\!]_3^{S'}([v_1 \mapsto t\_embed(u_1), v_2 \mapsto t\_embed(u_2)]) \\
 & = & [\![v_1 = v_2]\!]_3^{S'}([v_1 \mapsto u', v_2 \mapsto u']) \\
 & = & 1/2
\end{array}
$$

$$
\begin{array}{rcl}
1 & = & [\![v = v]\!]_3^S([v \mapsto u_1]) \\
 & \sqsubseteq & [\![v = v]\!]_3^{S'}(t\_embed\; o\, [v \mapsto u_1]) \\
 & = & [\![v = v]\!]_3^{S'}([v \mapsto t\_embed(u_1)]) \\
 & = & [\![v = v]\!]_3^{S'}([v \mapsto u']) \\
 & = & 1/2
\end{array}
$$

□

The Embedding Theorem requires that f be surjective in order to guarantee that an existential formula, such as $\exists v : \varphi$, has consistent values in S and S'. If f were not surjective, then there could exist an individual $u' \in U^{S'}$, not in the range of f, such that $[\![\varphi]\!]_3^{S'}([v \mapsto u']) = 1$. This would permit there to be structures S and S' for which $[\![\exists v : \varphi]\!]_3^S(Z) = 0$ but $[\![\exists v : \varphi]\!]_3^{S'}(f\; o\, Z) = 1$.

Apart from surjectivity, the Embedding Theorem depends on the fact that the three-valued meaning function is monotonic in its "assignment" argument (cf. Lemma 3.7).

As mentioned in the Introduction, one of the nice properties of Kleene's three-valued logic is that it coincides with two-valued logic on the two values 0 and 1. This is useful for shape analysis, because we wish to relate concrete (two-valued) structures and abstract (three-valued) structures. Furthermore, our methodology of expressing everything by means of formulae allows us to make a statement about both worlds via a single formula—the same syntactic expression can be interpreted with respect to either a two-valued structure or a three-valued structure. The Embedding Theorem (Theorem 3.11) gives us the tool to relate the two- and three-valued interpretations.

3.5 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 3, 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.

(Similar statements hold for the instrumentation predicates, as indicated in Table 4.)

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 6 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 (28) captures the fact that every program variable points to at most one list element. Formula (29) 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. Formula (30) means that all *sm* predicate values are 0 in

$$
\begin{array}{rcll}
\text{for each } \mathtt{x} \in PVar, \forall v_1, v_2 : x(v_1) \wedge x(v_2) & \Longrightarrow & v_1 = v_2 & (28)\\
\forall v_1, v_2 : (\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2)) & \Longrightarrow & v_1 = v_2 & (29)\\
 & & \forall v : \neg sm(v) & (30)\\
\forall v : (\exists v_1, v_2 : v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v)) & \Longrightarrow & is(v) & (31)\\
\forall v : (\forall v_1, v_2 : v_1 = v_2 \vee n(v_1, v) \vee n(v_2, v)) & \Longrightarrow & \neg is(v) & (32)\\
\hline
\text{for each } \mathtt{x} \in PVar, \forall v_2 : (\exists v_1 : v_1 \neq v_2 \wedge x(v_1)) & \Longrightarrow & \neg x(v_2) & (33)\\
\text{for each } \mathtt{x} \in PVar, \forall v_1 : (\exists v_2 : v_1 \neq v_2 \wedge x(v_2)) & \Longrightarrow & \neg x(v_1) & (34)\\
\forall v_2, v_3 : (\exists v_1 : v_1 \neq v_2 \wedge n(v_3, v_1)) & \Longrightarrow & \neg n(v_3, v_2) & (35)\\
\forall v_1, v_3 : (\exists v_2 : v_1 \neq v_2 \wedge n(v_3, v_2)) & \Longrightarrow & \neg n(v_3, v_1) & (36)\\
\forall v_2, v : (\exists v_1 : \neg is(v) \wedge v_1 \neq v_2 \wedge n(v_1, v)) & \Longrightarrow & \neg n(v_2, v) & (37)\\
\forall v_1, v : (\exists v_2 : \neg is(v) \wedge v_1 \neq v_2 \wedge n(v_1, v)) & \Longrightarrow & \neg n(v_1, v) & (38)\\
\forall v_1, v_2 : (\exists v : \neg is(v) \wedge n(v_1, v) \wedge n(v_2, v)) & \Longrightarrow & v_1 = v_2 & (39)
\end{array}
$$

Table 6: 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.

two-valued structures. The rules below the line are logical consequences of the rules above the line, and are generated systematically from them, as explained in Section 5.2.1.

In addition, for every instrumentation predicate $p \in \mathcal{I}$ defined by a formula $\varphi_p(v_1, v_2, \ldots, v_k)$, we generate a compatibility formula of the following form:

$$
\forall v_1, v_2, \ldots, v_k : \varphi_p(v_1, v_2, \ldots, v_k) \iff p(v_1, v_2, \ldots, v_k) \tag{40}
$$

This is then broken into two formulae of the form:

$$
\begin{array}{rcl}
\forall v_1, v_2, \ldots, v_k : \varphi_p(v_1, v_2, \ldots, v_k) & \Longrightarrow & p(v_1, v_2, \ldots, v_k)\\
\forall v_1, v_2, \ldots, v_k : \neg\varphi_p(v_1, v_2, \ldots, v_k) & \Longrightarrow & \neg p(v_1, v_2, \ldots, v_k)
\end{array}
$$

For instance, for the instrumentation predicate is, we use formula (4) for φ_{is} to generate compatibility formulae (31) and (32).

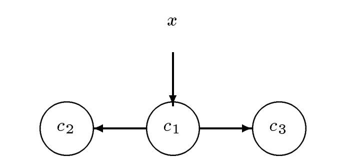
In the remainder of the paper, 2-CSTRUCT$[\mathcal{P}, F]$ denotes the set of two-valued structures that satisfy a set of compatibility formulae F.

We can exploit the close relationship between two-valued and three-valued logic to extend the hygiene conditions to three-valued structures. As with the two-valued structures 2-STRUCT$[\mathcal{P}]$, the set of three-valued structures 3-STRUCT$[\mathcal{P}]$ is more general than is necessary for shape analysis. One way to impose hygiene conditions on three-valued structures is merely to use the same set of compatibility formulae F that we use for two-valued structures, but to interpret the formulae in F under the three-valued interpretation (i.e., Definition 3.4). By the Embedding Theorem, this is safe: Because we are only concerned with two-valued structures $S^\natural \in$ 2-STRUCT$[\mathcal{P}]$ that *satisfy* all of the formulae in F, we need only be interested in three-valued structures $S \in$ 3-STRUCT$[\mathcal{P}]$ that *potentially satisfy* all of the formulae in F.

An alternative way to impose hygiene conditions on three-valued structures is developed 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



unary

el.	x	y	t	sm	is
c_1	1	0	0	0	0
c_2	0	0	0	0	0
c_3	0	0	0	0	0

n

	c_1	c_2	c_3
c_1	0	1	1
c_2	0	0	0
c_3	0	0	0

Figure 5: This structure, $S^{\natural}_{weird}$, is not represented by the structure S_2 from Figure 2.

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.,*

$$\gamma(S) = \{S^{\natural} \mid S^{\natural} \sqsubseteq S, S^{\natural} \in \textit{2-CSTRUCT}[\mathcal{P}, F]\} \tag{41}$$

□

Example 4.2 The structure S_2 shown in Figure 2 represents lists of length two or more. Notice that individual u of U^{S_2} also represents unreachable elements (i.e., uncollected garbage). Thus, this structure can actually represent a list with one element and one or more garbage cells. It is possible to change the definition of embeddings (abstractions) to explicitly exclude garbage cells (see [SRW96]). An alternative is to use an additional instrumentation predicate, r, defined by formula (5), to maintain reachability information explicitly. For every program statement, there would be a predicate-update formula to update r.

The structure $S^{\natural}_{weird}$ shown in Figure 5 has an individual c_1 that has two different outgoing `n` pointers. Because of the clause "$S^{\natural} \in$ 2-CSTRUCT$[\mathcal{P}, F]$" in the set-former in equation (41), S_2 does not represent the structure $S^{\natural}_{weird}$, even though $S^{\natural}_{weird} \sqsubseteq S_2$. □

4.2 The Meaning of Program Statements

The most technically challenging aspect in the design of our analysis is creating the abstract meaning functions for the program statements, which are defined as transformers from three-valued structures to three-valued structures. This task is difficult (even in a non-parametric framework) because of the following issues:

- It is hard to model the effect of program statements that destructively update memory locations, e.g., statements of the form `y->n = t`. Because of this, most pointer-analysis algorithms resort to imprecise approaches in many cases, such as performing weak updates (i.e., `n` edges emanating from the shape-node that `x` points to are accumulated) [LH88, CWZ90].
- The (three-valued) interpretation of different predicate symbols may be related. For example, heap sharing (i.e., predicate is) constrains the number of incoming selector edges (i.e., predicate n); conversely, the number of incoming selector edges constrains heap sharing.

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^{st}_p (referred to as p' in Section 2.3).

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 $\mathtt{z} \in PVar$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} (n(v_1, v_2) \wedge \neg x(v_1)) \vee (x(v_1) \wedge t(v_2))$ $\varphi_{sm}^{st}(v) \stackrel{\text{def}}{=} sm(v)$
`x = malloc()`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} new(v)$ $\varphi_z^{st}(v) \stackrel{\text{def}}{=} z(v) \wedge \neg new(v)$, for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} n(v_1, v_2) \wedge \neg new(v_1) \wedge \neg new(v_2)$ $\varphi_{sm}^{st}(v) \stackrel{\text{def}}{=} sm(v) \wedge \neg new(v)$

Table 7: Predicate-update formulae for the core predicates for `List` and `reverse`.

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 $v_1, v_2, \ldots, v_k$ that defines the new value of p after st. Then, the $\mathcal{P}$* **transformer associated with** *st, denoted by $[\![st]\!]$, is defined as follows:*

$$[\![st]\!](S) = \left\langle \begin{array}{l} U^S, \\ \lambda p.\lambda u_1, \ldots, u_k.[\![\varphi_p^{st}]\!]_3^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]) \end{array} \right\rangle \tag{42}$$

□

Example 4.4 Table 7 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, we need to use the modified definition of $[\![st]\!](S)$ given in Definition 4.5. For the statement `x = malloc()`, Definition 4.5 first allocates a new individual u_{new} and then invokes predicate-update formulae in a similar way to Definition 4.3.

Definition 4.5 *Let $st \equiv \mathtt{x} = \mathtt{malloc}()$ 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**

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

Table 8: Predicate-update formulae for the instrumentation predicate is.

with *$st \equiv$ `x = malloc()`, denoted by $[\![x = malloc()]\!]$, is defined as follows:*

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

□

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 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,*

$$[\![\varphi_p^{st}]\!]_3^{S^\natural}(Z) = [\![\varphi_p]\!]_3^{[\![st]\!](S^\natural)}(Z). \tag{43}$$

□

Example 4.7 Table 8 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 (43) holds. □

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

For three-valued structures, the analog of equation (43) does not hold, even for those structures that are tight embeddings of concrete structures from 2-CSTRUCT$[\mathcal{P}, F]$. For example, for the abstract execution of the first statement in Table 2 (st_1 : `y = NULL`), which transforms structure S_2 to S_3, we have $[\![\varphi_{is}]\!]_3^{S_3}([v \mapsto u]) = 1/2$, whereas the predicate-update formula yields $[\![\varphi_{is}^{st}]\!]_3^{S_2}([v \mapsto u]) = 0$. In fact, this is the origin of the Instrumentation Principle (Observation 2.2): The left-hand side can be more precise than the right-hand side (for tight embeddings of compatible structures).

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:

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, we denote the set of such structures by B-STRUCT[$\mathcal{P}$]. □

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

$$\begin{array}{rcl} \mathcal{C} & = & \{sm, n\} \cup \{x \mid \mathtt{x} \in PVar\} \\ \mathcal{I} & = & \{is\}. \end{array}$$

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 maximal number of individuals in a structure is $2^5 = 32$; however, because sm cannot have the value 1, the maximal number of individuals in a structure is really only 16. (On the other hand, Table 2 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. This is formalized in the following definition:

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 Table 2, the canonical name of u_1 is $u_{\{x,y\},\{t,is\}}$, and the canonical name of u is $u_{\emptyset,\{x,y,t,is\}}$. □

For any two bounded structures $S, S' \in$ B-STRUCT[$\mathcal{P}$], it is possible to check whether $S \sqsubseteq S'$ holds in time linear in the (explicit) sizes of S and S', using the following two-phase procedure:

1. Rename the individuals in U^S and $U^{S'}$ according to their canonical names.
2. For each predicate symbol $p \in \mathcal{P}$, check that the embedding condition (i.e., equation (24)) holds by comparing the predicates $\iota^S(p)$ and $\iota^{S'}(p)$.

If $S \sqsubseteq S'$ and $S' \sqsubseteq S$, then S and S' are isomorphic.

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, canonical abstraction is a generalization of the abstraction function used in [SRW98].[7] The only abstraction predicates used in [SRW98] are the "pointed-to-by-x" predicates. Because sharing predicate is is used only as an instrumentation predicate in [SRW98], but not as an abstraction predicate, the algorithm from [SRW98] does not distinguish between shared and unshared individuals, and thus loses accuracy for stores that contain a shared heap cell that is not directly pointed to by a variable. Adopting is as an additional abstraction predicate improves the accuracy of shape analysis: In this case, concrete-store elements that are shared and concrete-store elements that are not shared are represented by abstract individuals that have different canonical names.

Remark. The term "canonical abstraction" was chosen as a reminder that t_embed_c is a generalization of the abstraction functions that have been used in some of the previous work on shape analysis.

As mentioned in the Introduction, an idea that has been used in [Wan94, SRW98] to embed unbounded-size stores into bounded-size abstractions is to collapse concrete elements that are not directly pointed to by program variables into one abstract element, whereas concrete elements that are pointed to by different sets of variables are kept apart in different abstract elements. Earlier, Jones and Muchnick proposed making even finer distinctions by keeping exact information on elements within a distance k from a variable [JM81]. These collapsing processes can be generalized to any fixed set of unary "abstraction properties" on individuals:

> Individuals are partitioned into equivalence classes according to their sets of unary abstraction-property 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 method of collapsing structures always yields bounded structures, and is the **abstraction principle** behind canonical abstraction.

Compared to previous work, however, the present paper uses canonical abstraction in somewhat different ways:

- Because the concrete and abstract worlds are defined in terms of a single unified concept of three-valued logical structures, we are able to apply t_embed_c to abstract (three-valued) structures as well as to concrete (two-valued) ones.
- The present paper is not so tightly tied to canonical abstractions. There is nothing special about a bounded structure that uses canonical names; whenever necessary, canonical names can be recovered from the values of a structure's unary predicates.
- At various stages, we work with non-bounded structures, and return to bounded structures by applying t_embed_c. (The Embedding Theorem ensures that the operations we apply to three-valued structures are safe, even when we are working with non-bounded structures.)

□

The notion of bounded structures is very general, and not restricted to shape analysis. Many of the constructions presented in the rest of the paper are ones that apply to all classes of bounded structures, not just the ones we use in shape analysis.

4.4 The Shape-Analysis Algorithm

In this section, we define the actual shape-analysis algorithm. First, we define the Hoare order on sets of structures that is induced by the embedding order. (The shape-analysis algorithm is defined as a least-fixed point with respect to this order.)

[7] The shape-analysis algorithm presented in [SRW98] 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.

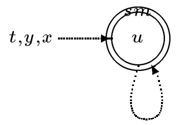
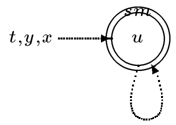
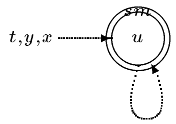
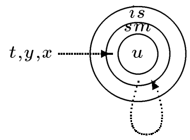
statement	formula	structure
st_2: `t = y;`	$t'(v) = y(v)$	S'_{12} 
st_3: `y = x;`	$y'(v) = x(v)$	S'_{13} 
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$	S'_{14} 
st_5: `y->n = t;`	$n'(v_1, v_2) = (n(v_1, v_2) \wedge \neg y(v_1)) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = \begin{array}{l} \exists v_1, v_2 : \\ (is(v) \wedge n'(v_1, v) \wedge n'(v_2, v) \wedge v_1 \neq v_2) \vee \\ (t(v) \wedge n(v_1, v) \wedge \neg y(v_1)) \end{array}$	S'_{15} 

Table 9: The bounded structures that actually arise for the last four blocks of Table 2 when t_embed_c is applied at each step.

Definition 4.12 *For sets of structures $XS_1, XS_1 \subseteq$ 3-STRUCT[$\mathcal{P}$], we define:*

$$XS_1 \sqsubseteq XS_2 \iff \forall S_1 \in XS_2 : \exists S_2 \in XS_2 : S_1 \sqsubseteq S_2 \tag{44}$$

□

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]:

$$\mathit{StructSet}[v] = \begin{cases} \{\langle \emptyset, \emptyset \rangle\} & \text{if } v = \mathit{start} \\ \bigcup_{v \to w \in G} \{t\_embed_c[\![st(w)]\!](S) \mid S \in \mathit{StructSet}[w]\} & \text{otherwise} \end{cases} \tag{45}$$

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.

In Section 2, we did not wish to complicate the discussion with the issue of mapping structures into bounded structures. For this reason, the last four blocks of Table 2 are deliberately inconsistent with equation (45). The bounded structures that actually arise when t_embed_c is applied at each step are shown in Table 9. In structures S'_{12}, S'_{13}, and S'_{14}, the canonical name of the one individual u is $u_{\emptyset,\{is\}}$. In structure S'_{15}, the canonical name of individual u is $u_{\emptyset,\emptyset}$.[8]

Other variations on equation (45) are possible. In particular, the function t_embed_c need not be applied after every statement; it could be applied (i) at every merge point in the control-flow graph, or (ii) only in loops, e.g., on each backedge of the control-flow graph.

[8]A consequence of Definition 4.8 is that when a bounded structure contains an individual whose canonical name is $u_{\emptyset,\emptyset}$, then this element is the only individual.

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

Figure 6: One- vs. three-stage abstract semantics of statement st_3. The operation $\llbracket st \rrbracket$ 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.)

5 Improved Abstract Semantics

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

This section is not organized to have a separate overview section for this material; however, the basic principles should be clear from Figure 7 and the corresponding examples.

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 6 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 $\llbracket st \rrbracket$, 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 $\llbracket st \rrbracket$) and may improve precision.

It is worthwhile noting that both *focus* and *coerce* are *semantic-reduction* operations (originally defined in [CC79]). 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 $\llbracket st \rrbracket$, 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. In Section 5.1.1, we define the *focus* operation, and in Section 5.1.2, we give an algorithm to actually compute *focus*.

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[$\mathcal{P}$],*

$$maximal(XS) \stackrel{\text{def}}{=} XS - \{X \in XS \mid \exists X' \in XS : X \sqsubseteq X' \textit{ and } X' \not\sqsubseteq X\} \tag{46}$$

$\square$

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 7 illustrates the application of *focus* to the formula $\varphi_x^{st_4}(v)$ and the structure S_5 that we have in `reverse` just after the first application of statement st_3: `y = x` and just before the first application of statement st_4: `x = x->n` in Table 2. 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 [SRW98]: 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 $S_{5,f,2}$. □

Remark. In the definition of *focus*, we apply *maximal* to reduce the number of structures that arise during shape analysis. □

The properties of *focus* are summarized in the following lemma:

Lemma 5.4 *For every pair of formulae φ_1 and φ_2 and structure S:*

Idempotence $focus_{\varphi_1}(focus_{\varphi_1}(S)) = focus_{\varphi_1}(S)$

Order-Independence $focus_{\varphi_1}(focus_{\varphi_2}(S)) = focus_{\varphi_2}(focus_{\varphi_1}(S))$

□

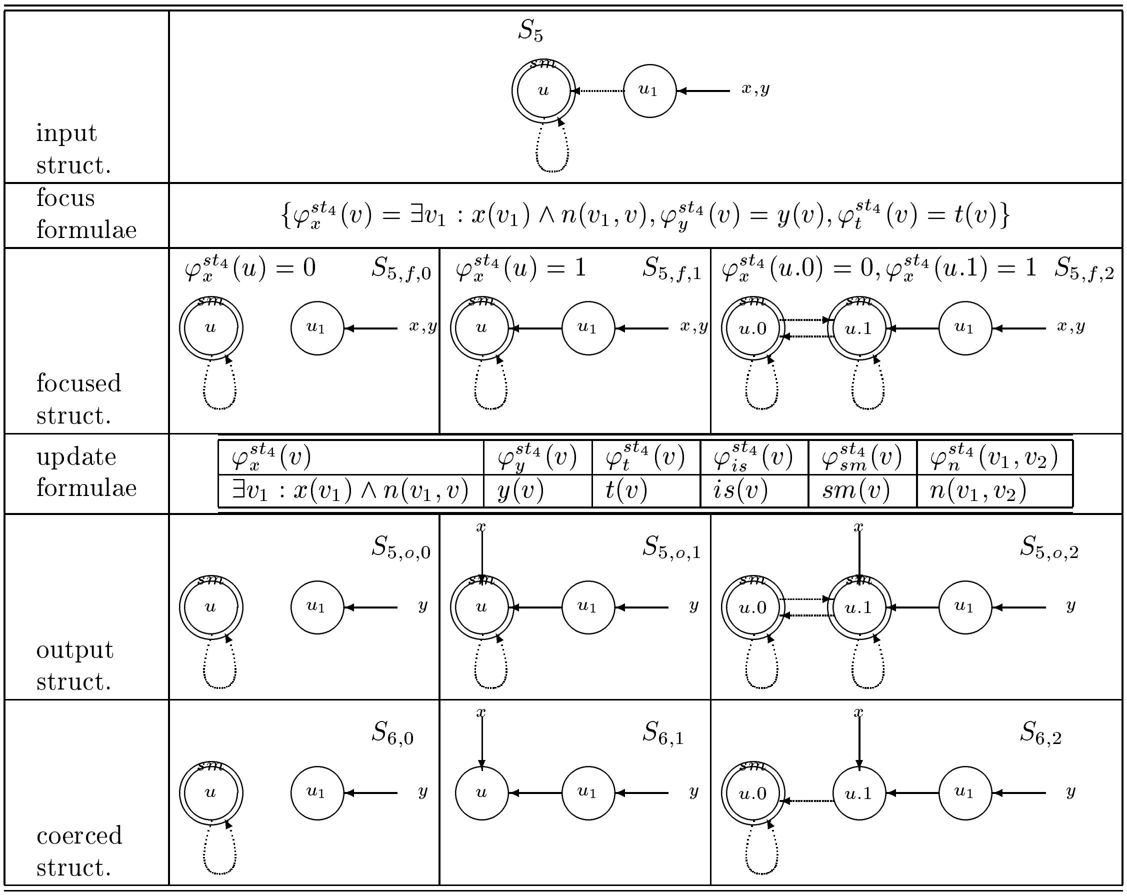


Figure 7: The first application of the improved transformer for statement st_4: `x = x->n` in `reverse`.

The fact that *focus* is order-independent allows us to extend it to work on sets of formulae by

$$focus_{\{\varphi_1, \varphi_2, \ldots, \varphi_l\}}(S) = focus_{\varphi_1}(focus_{\varphi_2}(\cdots(focus_{\varphi_l}(S)))). \tag{47}$$

The *focus* operation illustrated in Figure 7 is actually $focus_{\{\varphi_x^{st_4}, \varphi_y^{st_4}, \varphi_t^{st_4}\}}$, but in S_5 both $\varphi_y^{st_4}$ and $\varphi_t^{st_4}$ already have definite values for all of S_5's individuals.

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 5.5 *For every program variable* $\mathtt{x} \in PVar$*, statement* st*, and structure* S*,*

$$|focus_{\varphi_x^{st}}(S)| \leq 3|U^S|.$$

□

Example 5.6 In Example 5.3 and Table 7, there are three possible structures because indefinite values only occur in the summary node u. □

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 5.7 *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:*

$$
\begin{array}{rcl}
XS_1 \sqcup XS_2 & = & maximal(XS_1 \cup XS_2) \\
XS_1 \sqcap XS_2 & = & \left\{ \begin{array}{c} \langle U, \lambda p.\lambda u_1, \ldots u_k.\iota^{S_1}(p)(u_1, \ldots, u_k) \sqcap \iota^{S_2}(p)(u_1, \ldots, u_k)\rangle \\ \mid S_1 \in XS_1 \ and\ S_2 \in XS_2\ and\ comparable(S_1, S_2) \end{array} \right\}
\end{array}
$$

where:

$$
comparable(S_1, S_2) = \begin{array}{rl} & for\ all\ p \in \mathcal{P}, for\ all\ u_1, \ldots, u_k \in U: \\ & \iota^{S_1}(p)(u_1, \ldots, u_k) \sqsubseteq \iota^{S_2}(p)(u_1, \ldots, u_k) \\ or & \iota^{S_2}(p)(u_1, \ldots, u_k) \sqsubseteq \iota^{S_1}(p)(u_1, \ldots, u_k) \end{array} \tag{48}
$$

□

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 5.8 *Let $S = \langle U, \iota\rangle \in$ 3-STRUCT$[\mathcal{P}]$ be a three-valued structure. For a formula φ, and assignment Z, we define $z(\varphi)(S, Z), o(\varphi)(S, Z) \in 2^{\text{3-STRUCT}[\mathcal{P}]}$ inductively, as follows:*

$$
\begin{array}{rcl}
z(\boldsymbol{l})(S,Z) & = & \begin{cases} \{S\} & \boldsymbol{l} = \mathbf{0} \\ \emptyset & \text{otherwise} \end{cases} \\
z(v_1 = v_2)(S,Z) & = & \begin{cases} \{S\} & Z(v_1) \neq Z(v_2) \\ \emptyset & \text{otherwise} \end{cases} \\
z(p(v_1, \ldots, v_k))(S,Z) & = & \begin{cases} \{\langle U, \iota[p(Z(v_1), \ldots, Z(v_k)) \leftarrow 0]\rangle\} & 0 \sqsubseteq \iota(p(Z(v_1), \ldots, Z(v_k))) \\ \emptyset & \text{otherwise} \end{cases} \\
z(\varphi_1 \wedge \varphi_2)(S,Z) & = & z(\varphi_1)(S,Z) \sqcup z(\varphi_2)(S,Z) \\
z(\varphi_1 \vee \varphi_2)(S,Z) & = & z(\varphi_1)(S,Z) \sqcap z(\varphi_2)(S,Z) \\
z(\neg\varphi)(S,Z) & = & o(\varphi)(S,Z) \\
z(\forall v : \varphi)(S,Z) & = & \displaystyle\bigsqcup_{u \in U} z(\varphi)(S, Z[v \mapsto u]) \\
z(\exists v : \varphi)(S,Z) & = & \displaystyle\bigsqcap_{u \in U} z(\varphi)(S, Z[v \mapsto u])
\end{array}
$$

$$
\begin{array}{rcl}
o(\boldsymbol{l})(S,Z) & = & \begin{cases} \{S\} & \boldsymbol{l} = \mathbf{1} \\ \emptyset & \text{otherwise} \end{cases} \\
o(v_1 = v_2)(S,Z) & = & \begin{cases} z(sm(v_1))(S,Z) & Z(v_1) = Z(v_2) \\ \emptyset & \text{otherwise} \end{cases} \\
o(p(v_1, \ldots, v_k))(S,Z) & = & \begin{cases} \{\langle U, \iota[p(Z(v_1), \ldots, Z(v_k)) \leftarrow 1]\rangle\} & 1 \sqsubseteq \iota(p(Z(v_1), \ldots, Z(v_k))) \\ \emptyset & \text{otherwise} \end{cases} \\
o(\varphi_1 \wedge \varphi_2)(S,Z) & = & o(\varphi_1)(S,Z) \sqcap o(\varphi_2)(S,Z) \\
o(\varphi_1 \vee \varphi_2)(S,Z) & = & o(\varphi_1)(S,Z) \sqcup o(\varphi_2)(S,Z) \\
o(\neg\varphi)(S,Z) & = & z(\varphi)(S,Z) \\
o(\forall v : \varphi)(S,Z) & = & \displaystyle\bigsqcap_{u \in U} o(\varphi)(S, Z[v \mapsto u]) \\
o(\exists v : \varphi)(S,Z) & = & \displaystyle\bigsqcup_{u \in U} o(\varphi)(S, Z[v \mapsto u])
\end{array}
$$

□

The notation $\iota[p(u_1, \ldots, u_k) \leftarrow l]$ denotes the map obtained from ι by updating $\iota(p)(u_1, \ldots, u_k)$ to have the value l.

Example 5.9 For the formula $\varphi_x^{st_4}(v)$, structure S_5 from Figure 7, and individual $u \in U^{S_5}$, we have:

$$
\begin{array}{ll}
\multicolumn{2}{l}{z(\varphi_x^{st_4})(S_5,[v \to u])} \\
= & z(\exists v_1 : x(v_1) \wedge n(v_1,v))(S_5,[v \to u]) \\
= & \displaystyle\bigsqcap_{u' \in \{u,u_1\}} z(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u']) \\
= & z(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u]) \sqcap z(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u_1]) \\
= & \begin{array}{l}(z(x(v_1)(S_5,[v \to u, v_1 \to u]) \sqcup z(n(v_1,v))(S_5,[v \to u, v_1 \to u]))\sqcap \\ z(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u_1])\end{array} \\
= & \begin{array}{l}(\{S_5\} \sqcup \langle \{u,u_1\}, \iota^{S_5}[n(u,u) \mapsto 0]\rangle)\sqcap \\ z(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u_1])\end{array} \\
= & \begin{array}{l}\{S_5\}\sqcap \\ z(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u_1])\end{array} \\
= & \begin{array}{l}\{S_5\}\sqcap \\ (z(x(v_1))(S_5,[v \to u, v_1 \to u_1]) \sqcup z(n(v_1,v))(S_5,[v \to u, v_1 \to u_1]))\end{array} \\
= & \begin{array}{l}\{S_5\}\sqcap \\ (\emptyset \sqcup z(n(v_1,v))(S_5,[v \to u, v_1 \to u_1]))\end{array} \\
= & \{S_5\} \sqcap z(n(v_1,v))(S_5,[v \to u, v_1 \to u_1]) \\
= & \{S_5\} \sqcap \langle \{u,u_1\}, \iota^{S_5}[n(u_1,u) \mapsto 0]\rangle \\
= & \{\langle \{u,u_1\}, \iota^{S_5}[n(u_1,u) \mapsto 0]\rangle\} \\
= & \{S_{5,f,0}\}
\end{array}
$$

and

$$
\begin{array}{ll}
\multicolumn{2}{l}{o(\varphi_x^{st_4})(S_5,[v \to u])} \\
= & o(\exists v_1 : x(v_1) \wedge n(v_1,v))(S_5,[v \to u]) \\
= & \displaystyle\bigsqcup_{u' \in \{u,u_1\}} z(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u']) \\
= & o(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u]) \sqcup o(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u_1]) \\
= & \begin{array}{l}(o(x(v_1)(S_5,[v \to u, v_1 \to u]) \sqcap o(n(v_1,v))(S_5,[v \to u, v_1 \to u]))\sqcup \\ o(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u_1])\end{array} \\
= & \begin{array}{l}(\emptyset \sqcap o(n(v_1,v))(S_5,[v \to u, v_1 \to u]))\sqcup \\ o(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u_1])\end{array} \\
= & \emptyset \sqcup o(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u_1]) \\
= & o(x(v_1) \wedge n(v_1,v))(S_5,[v \to u, v_1 \to u_1]) \\
= & o(x(v_1))(S_5,[v \to u, v_1 \to u_1]) \sqcap o(n(v_1,v))(S_5,[v \to u, v_1 \to u_1]) \\
= & \{S_5\} \sqcap \{\langle \{u,u_1\}, \iota^{S_5}[n(u_1,u) \mapsto 1]\rangle\} \\
= & \{S_{5,f,1}\}
\end{array}
$$

□

Remark. In Definition 5.8 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. For example, $o(p^+(v_1,v_2))(S,Z)$ would yield the set of maximal structures that can be embedded in S, and in which there exists a definite p-path from $Z(v_1)$ to $Z(v_2)$. □

The algorithm for *focus*, called Focus, is shown in Figure 8. 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. [SRW98]).

Example 5.10 Consider the application of Focus to the structure S_5 from Figure 7 and the formula $\varphi_x^{st_4}$. By Example 5.9, $z(\varphi_x^{st_4})(S_5,Z)$ 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 $\{S_{5,f,0}, S_{5,f,1}, S_{5,f,2}\}$. □

function Focus(S : 3-STRUCT[$\mathcal{P}$], $\varphi^{st}_x(v)$: Formula) **returns** $2^{\text{3-CSTRUCT}[\mathcal{P},R(F)]}$
begin
  if there exists $u \in U^S$ such that $[\![\varphi^{st}_x]\!]^S_3([v \mapsto u]) = 1/2$ **then**
    let $u.0$ and $u.1$ be individuals not in U^S
    and $XS = \begin{array}{ll} & z(\varphi^{st}_x(v))(S, [v \mapsto u]) \\ \cup & o(\varphi^{st}_x(v))(S, [v \mapsto u]) \\ \cup & o(\varphi^{st}_x(v))(z(\varphi^{st}_x(v))(\text{Expand}(S, u, u.0, u.1), [v \mapsto u.0]), [v \mapsto u.1]) \end{array}$
    in return $\bigcup_{S' \in XS} \text{Focus}(S', \varphi^{st}_x(v))$
  else return $\{S\}$
end

function Expand(S : 3-STRUCT[$\mathcal{P}$], $u, u.0, u.1$: elements) **returns** 3-STRUCT[$\mathcal{P}$]
let $m = \lambda u'.\begin{cases} u & u' = u.0 \vee u' = u.1 \\ u' & \text{otherwise} \end{cases}$ **in**
  return $\left\langle \begin{array}{l} (U^S - \{u\}) \cup \{u.0, u.1\} \\ \lambda p.\lambda u_1, u_2, \ldots, u_k.\iota^S(p)(m(u_1), m(u_2), \ldots, m(u_k)) \end{array} \right\rangle$

Figure 8: An algorithm for $focus_{\varphi^{st}_x(v)}$.

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 5.11 Consider structure $S_{5,o,2}$ from Figure 7. 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 $u.0$. $\square$

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 5.11. Furthermore, the shape-analysis actually yields precise information in the analysis of `reverse`.

This subsection is organized as follows: In Section 5.2.1, we show that structures that result from abstraction obey certain consistency rules. In particular, the Property-Extraction and Instrumentation Principles play important roles here. Interestingly, the consistency rules are stronger than what we have when the formulae for hygiene conditions on two-valued structures are just interpreted as three-valued formulae. The consistency rules are used in Section 5.2.2 to define an operation, called *coerce*, that "coerces" a structure into a more precise structure. Finally, in Section 5.2.3, we give an algorithm for *coerce*.

5.2.1 Compatibility Constraints

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

Example 5.12 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 3.8) 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 3.9—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 5.12 can be summarized as the following principle:

Observation 5.13 [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., $[\![\varphi_p]\!]_3^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k])$). 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.* □

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

Definition 5.14 *Let Σ be a finite set of compatibility constraints of the form:*

$$\varphi_1 \triangleright \varphi_2 \tag{49}$$

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 in Σ of the form (49), and for every assignment Z such that $[\![\varphi_1]\!]_3^S(Z) = 1$, we have $[\![\varphi_2]\!]_3^S(Z) = 1$.* □

For a two-valued structure, $\triangleright$ has the same meaning as implication. However, for a three-valued structure $\triangleright$ is stronger than implication: if φ_1 evaluates to 1 and φ_2 evaluates to 1/2, the constraint $\varphi_1 \triangleright \varphi_2$ is not satisfied. More precisely, suppose that $[\![\varphi_1]\!]_3^S(Z) = 1$ and $[\![\varphi_2]\!]_3^S(Z) = 1/2$; the implication $\varphi_1 \implies \varphi_2$ is satisfied (i.e., $S, Z \models \varphi_1 \implies \varphi_2$), but the constraint $\varphi_1 \triangleright \varphi_2$ is not satisfied (i.e., $S, Z \not\models \varphi_1 \triangleright \varphi_2$).

The constraint that captures the reasoning used in Example 5.12 is

$$\varphi_{is}(v) \triangleright is(v)$$

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.

Constraints give us a way to express certain properties that are a consequence of the tight-embedding process, but that would not be expressible with formulae alone. In general, constraints are not expressible in Kleene's logic (i.e., by means of a formula that simulates the connective $\triangleright$). The reason is that formulae are monotonic in the information order (see Lemma 3.7), whereas $\triangleright$ is non-monotonic in its right-hand-side argument: The constraint $\mathbf{1} \triangleright \mathbf{1}$ is satisfied in all structures; however, when we change the right-hand-side argument from $\mathbf{1}$ to $\mathbf{1/2}$, we find that the constraint $\mathbf{1} \triangleright \mathbf{1/2}$ is one that is not satisfied in any structure.

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

$$
\begin{array}{rcll}
\text{for each } \mathtt{x} \in PVar, x(v_1) \wedge x(v_2) & \triangleright & v_1 = v_2 & (53)\\
(\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2)) & \triangleright & v_1 = v_2 & (54)\\
(\exists v_1, v_2 : v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v)) & \triangleright & is(v) & (55)\\
(\forall v_1, v_2 : v_1 = v_2 \vee \neg n(v_1, v) \vee \neg n(v_2, v)) & \triangleright & \neg is(v) & (56)\\
sm(v) & \triangleright & \mathbf{0} & (57)\\
\hline
\text{for each } \mathtt{x} \in PVar, (\exists v_1 : v_1 \neq v_2 \wedge x(v_1)) & \triangleright & \neg x(v_2) & (58)\\
\text{for each } \mathtt{x} \in PVar, (\exists v_2 : v_1 \neq v_2 \wedge x(v_2)) & \triangleright & \neg x(v_1) & (59)\\
(\exists v_1 : v_1 \neq v_2 \wedge n(v_3, v_1)) & \triangleright & \neg n(v_3, v_2) & (60)\\
(\exists v_2 : v_1 \neq v_2 \wedge n(v_3, v_2)) & \triangleright & \neg n(v_3, v_1) & (61)\\
(\exists v_1 : \neg is(v) \wedge v_1 \neq v_2 \wedge n(v_1, v)) & \triangleright & \neg n(v_2, v) & (62)\\
(\exists v_2 : \neg is(v) \wedge v_1 \neq v_2 \wedge n(v_2, v)) & \triangleright & \neg n(v_1, v) & (63)\\
(\exists v : \neg is(v) \wedge n(v_1, v) \wedge n(v_2, v)) & \triangleright & v_1 = v_2 & (64)
\end{array}
$$

Table 10: The compatibility constraints $R(\widehat{closure}(F))$ generated using Definitions 5.15 and 5.17 from the formulae F given above the line in Table 6. The constraints below the line come from applying r to the formulae listed below the line in Table 6.

Definition 5.15 *For formula φ and atomic formula a, define $r(\varphi)$ as follows.*

$$
\begin{aligned}
r(\forall v_1, v_2, \ldots v_k : (\varphi \implies a)) &\stackrel{\text{def}}{=} \varphi \triangleright a && (50)\\
r(\forall v_1, v_2, \ldots v_k : (\varphi \implies \neg a)) &\stackrel{\text{def}}{=} \varphi \triangleright \neg a && (51)\\
r(\forall v_1, v_2, \ldots v_k : \varphi) &\stackrel{\text{def}}{=} \neg\varphi \triangleright \mathbf{0} && (52)
\end{aligned}
$$

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

Example 5.16 The constraints generated for the formulae that appear above the line in Table 6 are listed above the line in Table 10. □

The constraint-generation rules defined in Definition 5.15 generate interesting constraints only for certain specific syntactic forms, namely implications with exactly one (possibly negated) predicate symbol on the right-hand side. Thus, when we generate compatibility constraints from implicative hygiene conditions (cf. Table 6 and the discussion in Section 3.5), the set of constraints generated depends on the form in which the hygiene conditions are written. In particular, not all of the many equivalent forms possible for a given hygiene condition lead to useful constraints. For instance, $r(\forall v_1, v_2, \ldots v_k : (\varphi \implies a))$ yields the (useful) constraint $\varphi \triangleright a$, but $r(\forall v_1, v_2, \ldots v_k : (\neg\varphi \vee a))$ yields the (not useful) constraint $\neg(\neg\varphi \vee a) \triangleright \mathbf{0}$.

This phenomenon can lead to a shape-analysis algorithm that is more conservative than we would like. However, when hygiene conditions are written as "extended Horn clauses" (see Definition 5.17 below), the way around this difficulty is to augment the constraint-generation process to generate constraints for some of the logical consequences of each hygiene condition. The process of "generating some of the logical consequences for extended Horn clauses" is formalized in the following definition:

Definition 5.17 *For a formula φ, we define $\varphi^1 \equiv \varphi$ and $\varphi^0 \equiv \neg\varphi$. For a formula φ of the form*

$$\forall \ldots : \bigvee_{i=1}^{m} (\varphi_i)^{B_i}$$

where $m > 1$ and $B_i \in \{0, 1\}$ (i.e., an **extended Horn clause***), we define closure(φ) to be the* *following set of formulae:*

$$closure(\varphi) \stackrel{\text{def}}{=} \left\{ \forall \ldots, \exists v_1, v_2, \ldots, v_n : \bigwedge_{i=1, i\neq j}^{m} \varphi_i^{1-B_i} \implies \varphi_j^{B_j} \;\middle|\; \begin{array}{l} 1 \leq j \leq m, \\ v_i \in \mathit{freeVars}(\varphi) - \mathit{freeVars}(\varphi_j) \end{array} \right\} \tag{65}$$

and otherwise $closure(\varphi) = \{\varphi\}$. *Finally, for a set of formulae* F*, we write* $\widehat{closure}(F)$ *to denote the application of closure to every formula in* F. □

It is easy to see that the formulae in $closure(\varphi)$ are implied by φ.

Example 5.18 Table 10 shows the compatibility constraints obtained from $R(\widehat{closure}(F))$, where F is the set of formulae given above the line in Table 6. For the extended Horn clauses that appear above the line in Table 6, the new formulae generated by *closure* are listed below the line in Table 6. The constraints generated from these formulae are listed below the line in Table 10.

As we will see in Section 5.2, compatibility constraints—and, in particular, the ones created from formulae generated by *closure*—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 (62) 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 (62) 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 5.11 and 5.23). 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 7), 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 5.19 (Compatible Three-Valued Structures). *The* **set of compatible three-valued structures** *3-CSTRUCT*$[\mathcal{P}, R(F)] \subseteq$ *3-STRUCT*$[\mathcal{P}]$ *is defined by* $S \in$ *3-CSTRUCT*$[\mathcal{P}, R(F)]$ *iff* $S \models R(F)$. □

The following lemma ensures that we can always replace a structure by a compatible one that satisfies constraint-set $R(F)$ without losing information:

Lemma 5.20 *For every structure* $S \in$ *3-STRUCT*$[\mathcal{P}]$ *and concrete structure* $S^\natural \in \gamma(S)$*, there exists a structure* $S' \in$ *3-CSTRUCT*$[\mathcal{P}, R(F)]$ *such that (i)* $U^{S'} = U^S$*, (ii)* $S' \sqsubseteq S$ *and (iii)* $S^\natural \in \gamma(S')$

Sketch of Proof: Let $S^\natural \in \gamma(S)$, then by Definition 4.1, $S^\natural \models F$ and there exists a function $f : U^{S^\natural} \to U^S$ such that $S^\natural \sqsubseteq^f S$. Define $S' = f(S^\natural)$ (i.e., S' is the tight embedding of $S^\natural$ under f). It is not hard to see that S' satisfies the necessary requirements. □

In Section 5.2.3, we give an algorithm that constructs from S and $R(F)$ a maximal S' meeting the conditions of Lemma 5.20 (without investigating the possibly infinite set of actual concrete structures $S^\natural \in \gamma(S)$).

5.2.2 The Coerce Operation

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

Example 5.21 Consider structure $S_{5,o,2}$ from Figure 7 again. The structure $S_{5,o,2}$ violates constraint (62) 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 (62) 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 5.22 *The operation coerce*: *3-STRUCT*$[\mathcal{P}] \to$ *3-CSTRUCT*$[\mathcal{P}, R(F)] \cup \{\bot\}$ *is defined as follows:*

$$coerce(S) \stackrel{\text{def}}{=} \begin{cases} \textit{the maximal } S' \textit{ s.t.} & (S' \sqsubseteq S) \textit{ and } (U^{S'} = U^S) \textit{ and } (S' \in \textit{3-CSTRUCT}[\mathcal{P}, R(F)]) \\ \bot & \textit{if no such } S' \textit{ exists} \end{cases}$$

□

It is not hard to see that Lemma 5.20 implies that the maximal such structure S' is unique (if it exists). Therefore, in the above definition, *coerce* is not defined in terms of a set former (unlike *focus*, which can return a non-singleton set).

Example 5.23 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 7. It yields $S_{6,0}, S_{6,1}$, and $S_{6,2}$, respectively.

- The structure $S_{6,0}$ is equal to $S_{5,o,0}$ because $S_{5,o,0}$ already satisfies all of the constraints of Figure 10.
- The structure $S_{6,1}$ was obtained from $S_{5,o,1}$ by removing incompatibilities as follows:
  1. Consider the assignment $[v \mapsto u, v_1 \mapsto u_1, v_2 \mapsto u]$. Because $\iota(is)(u) = 0$, $u_1 \neq u$, and $\iota(n)(u_1, u) = 1$, constraint (62) implies that $\iota(n)(u, u)$ must equal 0. Thus, in $S_{6,1}$ the (indefinite) `n` edge from u to u has been removed.
  2. Consider the assignment $[v_1 \mapsto u, v_2 \mapsto u]$. Because $\iota(x)(u) = 1$, constraint (53) implies that $[\![v_1 = v_2]\!]_3^{S_{6,1}}([v_1 \mapsto u, v_2 \mapsto u]) = 1$. By Definition 3.4, this means that $\iota(sm)(u)$ must equal 0. Thus, in $S_{6,1}$ u is no longer a summary node.
- The structure $S_{6,2}$ was obtained from $S_{5,o,2}$ by removing incompatibilities as follows:
  1. Consider 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 $\iota(n)(u_1, u.1) = 1$, constraint (62) implies that $\iota(n)(u.0, u.1)$ must equal 0. Thus, in $S_{6,2}$ the (indefinite) `n` edge from $u.0$ to $u.1$ has been removed.
  2. Consider the assignment $[v \mapsto u.1, v_1 \mapsto u_1, v_2 \mapsto u.1]$. Because $\iota(is)(u.1) = 0$, $u_1 \neq u.1$, and $\iota(n)(u_1, u.1) = 1$, constraint (62) implies that $\iota(n)(u.1, u.1)$ must equal 0. Thus, in $S_{6,2}$ the (indefinite) `n` edge from $u.1$ to $u.1$ has been removed.
  3. Consider the assignment $[v_1 \mapsto u.1, v_2 \mapsto u.1]$. Because $\iota(x)(u.1) = 1$, constraint (53) implies that $[\![v_1 = v_2]\!]_3^{S_{6,2}}([v_1 \mapsto u.1, v_2 \mapsto u.1]) = 1$. By Definition 3.4, this means that $\iota(sm)(u.1)$ must equal 0. Thus, in $S_{6,2}$ $u.1$ is no longer a summary node.

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 Table 2): 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 5.14. It is convenient to partition these constraints into the following types:

$$\begin{array}{lll} \varphi(v_1, v_2, \ldots, v_k) & \triangleright & \boldsymbol{b} \\ \varphi(v_1, v_2, \ldots, v_k) & \triangleright & (v_1 = v_2)^b \\ \varphi(v_1, v_2, \ldots, v_k) & \triangleright & p^b(v_1, v_2, \ldots, v_k) \end{array} \qquad \begin{array}{r} (66) \\ (67) \\ (68) \end{array}$$

where $\boldsymbol{b} \in \{\mathbf{0}, \mathbf{1}, \mathbf{1/2}\}$ and the superscript notation used is the same as in Definition 5.17: $\varphi^1 \equiv \varphi$ and $\varphi^0 \equiv \neg\varphi$. We say that constraints in the forms (66), (67), and (68) are *Type I*, *Type II*, and *Type III* constraints, respectively.

```
function Coerce(S: 3-STRUCT[P], R(F): Set of constraints)
returns 3-CSTRUCT[P, R(F)] ∪ {⊥}
begin
   S' := S
   while there exists a constraint r ≡ φ1 ▷ φ2 ∈ R(F) and an
      assignment Z: freeVars(r) → U^S such that S', Z ⊭ r do
      switch φ2
         case φ2 ≡ b /* Type I */
            return ⊥
         case φ2 ≡ (v1 = v2)^b /* Type II */
            if b = 1 and Z(v1) = Z(v2) and ι^{S'}(sm)(Z(v1)) = 1/2 then
              ι^{S'}(sm)(Z(v1)) := 0
            else return ⊥
         case φ2 ≡ p^b(v1, ..., vk) /* Type III */
            if ι^{S'}(p)(Z(v1), ..., Z(vk)) = 1/2 then
              ι^{S'}(p)(Z(v1), ..., Z(vk)) := b
            else return ⊥
      end switch
   od
   return S'
end
```

Figure 9: An iterative algorithm for solving three-valued constraints.

The algorithm for *coerce* is shown in Figure 9. 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 $\perp$, 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 5.24 *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 $\sum_{p \in \mathcal{P}} |U|^{arity(p)}$.

6 Instrumentation Predicates that Track Reachability Properties

One of the attractive features of having a parametric framework for shape analysis is the ability to define new shape-analysis algorithms easily by instantiating the framework using different collections of instrumentation predicates. This section discusses issues associated with two particularly important cases, both of which use instrumentation predicates that track reachability properties. The instrumentation predicates capture the following conditions:

- Is a cell reachable from a *specific* variable of the program?
- Is a cell reachable from *any* of the variables of the program?

These are discussed in Sections 6.1 and 6.2, respectively.

6.1 Reachability From Individual Program Variables

By using instrumentation predicates that track information about reachability from the individual variables of the program, shape-analysis algorithms can be created that, in many cases, determine precise shape information for programs that manipulate several (possibly cyclic) data structures simultaneously. The information obtained is more precise than that obtained from previous work on shape analysis [JM81, JM82, LH88, HPR89, CWZ90, Hen90, HN90, LR91, Str92, Deu92, AW93, PCK93, Deu94, Wan94, SRW98].

The definitions of the "reachable-from-`x`" predicates, denoted by r_x, were given in Section 3.1 as equation (6):

$$\varphi_{r_x}(v) \stackrel{\text{def}}{=} x(v_1) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v), \text{ for each } \mathtt{x} \in PVar.$$

Instrumentation-predicate $r_x(v)$ tracks the property "Is v reachable from pointer variable `x`?" These instrumentation predicates drastically improve the precision of shape analysis, even for programs that manipulate simple list and tree data structures, because they keep separate the abstract representations of data structures that are disjoint in the concrete world [SRW98, p.38].

In addition to supplying the definition of formula $\varphi_{r_x}(v)$, in order to instantiate the shape-analysis framework with the r_x predicates, it is necessary to supply predicate-update formulae that specify how each kind of statement causes the value of r_x to change. For programs that use the `List` data-type declaration from Figure 1(a), suitable predicate-update formulae are listed in Table 11. With the exception of the formulae for statements of the form `x->n = t`, it is not difficult to see that for every statement kind, the formulae in Table 11 maintain a correct instrumentation for the "reachable-from-`x`" predicates (i.e., equation (43) holds for $\varphi^{st}_{r_x}$ and φ_{r_x}, for all $\mathtt{x} \in PVar$).

In the predicate-update formulae in Table 11 for statements of the form `x->n = t`, the notation $\varphi_{r_{x \to n}}(v)$ is a shorthand for a formula that expresses "Is v reachable from node `x->n`?":

$$\varphi_{r_{x \to n}}(v) \stackrel{\text{def}}{=} \exists v_1, v_2 : x(v_1) \wedge n(v_1, v_2) \wedge ((v_2 = v) \vee n^+(v_2, v)).$$

When examining the predicate-update formulae for statements of the form `x->n = t`, it helps to keep in mind the following readings of the formulae listed below:

Term	Significance
$\varphi_{r_{x \to n}}(v)$	Is v reachable from node `x->n`?
$z(v) \vee \exists v_1 : z(v_1) \wedge (TC\ v_2, v_3 : \varphi^{st}_n(v_2, v_3))(v_1, v)$	Is v reachable from `z` using the updated `n` predicate?

Thus, the predicate-update formula $\varphi^{st}_{r_x}(v)$ given in Table 11 for variable `x` means the following (where "old store" means the store that we have before the execution of `x->n = t`, and "new store" means the store that arises after the execution of `x->n = t`):

- For an element u that is not reachable from `x->n` in the old store, u is reachable from `x` in the new store if and only if `x` points to u directly or u is reachable from `t` in the old store.

st	$\varphi^{st}_{r_x}$
`x = NULL`	$\varphi^{st}_{r_x}(v) \stackrel{\text{def}}{=} \mathbf{0}$ $\varphi^{st}_{r_z}(v) \stackrel{\text{def}}{=} r_z(v)$ for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$
`x = t`	$\varphi^{st}_{r_x}(v) \stackrel{\text{def}}{=} r_t(v)$ $\varphi^{st}_{r_z}(v) \stackrel{\text{def}}{=} r_z(v)$ for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$
`x = t->n`	$\varphi^{st}_{r_x}(v) \stackrel{\text{def}}{=} \exists v_1 : r_t(v_1) \wedge n(v_1, v)$ $\varphi^{st}_{r_z}(v) \stackrel{\text{def}}{=} r_z(v)$ for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$
`x->n = t`	$\varphi^{st}_{r_x}(v) \stackrel{\text{def}}{=} \begin{array}{l} (\neg\varphi_{r_{x\to n}}(v) \wedge (x(v) \vee r_t(v))) \\ \vee\ (\varphi_{r_{x\to n}}(v) \wedge (x(v) \vee \exists v_1 : x(v_1) \wedge (TC\ v_2, v_3 : \varphi^{st}_n(v_2, v_3))(v_1, v))) \end{array}$ $\varphi^{st}_{r_z}(v) \stackrel{\text{def}}{=} \begin{array}{l} (\neg\varphi_{r_{x\to n}}(v) \wedge r_z(v)) \\ \vee\ (\varphi_{r_{x\to n}}(v) \wedge (z(v) \vee \exists v_1 : z(v_1) \wedge (TC\ v_2, v_3 : \varphi^{st}_n(v_2, v_3))(v_1, v))) \end{array}$ for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$
`x = malloc()`	$\varphi^{st}_{r_x}(v) \stackrel{\text{def}}{=} new(v)$ $\varphi^{st}_{r_z}(v) \stackrel{\text{def}}{=} r_z(v) \wedge \neg new(v)$ for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$

Table 11: Predicate-update formulae for the instrumentation predicate r_x, for programs that use the `List` data-type declaration from Figure 1(a). The notation $\varphi_{r_{x\to n}}(v)$ is a shorthand for a formula that expresses "Is v reachable from node `x->n`?": It denotes the formula $\exists v_1, v_2 : x(v_1) \wedge n(v_1, v_2) \wedge ((v_2 = v) \vee n^+(v_2, v))$.

- For an element u that is reachable from `x->n` in the old store, the updated `n` predicate is used to determine whether u is reachable from `x` in the new store.

The predicate-update formula $\varphi^{st}_{r_z}(v)$ for any other variable `z`—where `z` could be `t`—means:

- For an element u that is not reachable from `x->n` in the old store, u is reachable from `z` in the new store if and only if u is reachable from `z` in the old store.
- For an element u that is reachable from `x->n` in the old store, the updated `n` predicate is used to determine whether u is reachable from `z` in the new store.

6.2 Reachability From Program Variables

This section discusses the use of a single instrumentation predicate that tracks information about whether a cell is reachable from any of the program's variables. The definition of this instrumentation predicate, denoted by $r(v)$, was given in Section 3.1 as equation (5):

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

It tracks the property "Is v reachable from *some* pointer variable?", or alternatively, "Is v a non-garbage element?"

The effect of instrumentation predicate r on shape descriptions is that garbage cells are represented by different individuals from non-garbage cells: Definite values of reachability predicate r distinguish non-garbage cells from garbage cells. (Without r, this distinction is not maintained. For instance, in an instantiation of the shape-analysis framework that does not use instrumentation predicate r, the structure S_2 shown in Figure 2 represents a store containing a list of length two or more, *or* a list of length one together with one or more garbage cells.) Information obtained from a shape-analysis algorithm that uses r could be used by an optimizing compiler to insert instructions in a program to perform compile-time garbage collection.

Because $\varphi_r(v)$ is just the disjunction of all of the φ_{r_x} formulae, for each program statement st, the predicate-update formula for r can be defined as the disjunction of the predicate-update formulae for the $\varphi^{st}_{r_x}$ predicates:

$$\varphi^{st}_r(v) \stackrel{\text{def}}{=} \bigvee_{\mathtt{x} \in PVar} \varphi^{st}_{r_x}(v).$$

Acknowledgements

We are grateful for the helpful comments of A. Avron, T. Ball, M. Benedikt, M. Gitik, K. Kunen, V. Lifschitz, H.R. Nielson, M. O'Donnell, A. Rabinovich, and K. Sieber.

A Proof of the Embedding Theorem

By the De Morgan laws it is sufficient to show the theorem for formulae involving $\wedge$, $\neg$, and $\exists$. The proof works by structural induction on φ:
Basis: For atomic formula $p(v_1, v_2, \ldots, v_k)$, $u_1, u_2, \ldots, u_k \in U^S$, and $Z = [v_1 \mapsto u_1, v_2 \mapsto u_2, \ldots, v_k \mapsto u_k]$ we have

$$\begin{array}{ll} [\![p(v_1, v_2, \ldots, v_k)]\!]^S_3(Z) & \\ \quad = \iota^S(p)(u_1, u_2, \ldots, u_k) & \text{(Definition 3.4)} \\ \quad \sqsubseteq \iota^{S'}(p)(f(u_1), f(u_2), \ldots, f(u_k)) & \text{(Definition 3.8)} \\ \quad = [\![p(v_1, v_2, \ldots, v_k)]\!]^{S'}_3(f \; o \; Z) & \text{(Definition 3.4)} \end{array}$$

Also for $l \in \{0, 1/2, 1\}$, we have:

$$\begin{array}{ll} [\![l]\!]^S_3(Z) & \\ \quad = l & \text{(Definition 3.4)} \\ \quad \sqsubseteq l & \text{(Definition 3.1)} \\ \quad = [\![l]\!]^{S'}_3(f \; o \; Z) & \text{(Definition 3.4)} \end{array}$$

Let us show that

$$[\![v_1 = v_2]\!]^S_3(Z) \sqsubseteq [\![v_1 = v_2]\!]^{S'}_3(f \; o \; Z)$$

If $[\![v_1 = v_2]\!]^{S'}_3(f \; o \; Z) = 1/2$ then the theorem holds for $v_1 = v_2$, trivially. If $[\![v_1 = v_2]\!]^{S'}_3(f \; o \; Z) = 1$ then by Definition 3.4, (i) $f(Z(v_1)) = f(Z(v_2))$ and (ii) $\iota^{S'}(sm)(f(Z(v_1))) = 0$. Therefore, by Definition 3.8, $Z(v_1) = Z(v_2)$. Hence, by Definition 3.4, $[\![v_1 = v_2]\!]^S_3(Z) = 1$. Finally, assume that $[\![v_1 = v_2]\!]^{S'}_3(f \; o \; Z) = 0$. In this case, by Definition 3.4, $f(Z(v_1)) \neq f(Z(v_2))$. Therefore, $Z(v_1) \neq Z(v_2)$. Therefore, by Definition 3.4 $[\![v_1 = v_2]\!]^S_3(Z) = 0$.
Induction step: Suppose φ is a formula with free variables $v_1, v_2, \ldots v_k$. Let Z be a complete assignment for φ. If $[\![\varphi]\!]^{S'}_3(Z) = 1/2$, then the theorem holds trivially. Therefore assume that $[\![\varphi]\!]^{S'}_3(f \; o \; Z) \in \{0, 1\}$. We distinguish between the following cases:

Logical-and $\varphi = \varphi_1 \wedge \varphi_2$. The proof splits into the following subcases:

Case 1: $[\![\varphi_1 \wedge \varphi_2]\!]^{S'}_3(f \; o \; Z) = 0$.
In this case, either $[\![\varphi_1]\!]^{S'}_3(f \; o \; Z) = 0$ or $[\![\varphi_2]\!]^{S'}_3(f \; o \; Z) = 0$. Without loss of generality assume that $[\![\varphi_1]\!]^{S'}_3(f \; o \; Z) = 0$. Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]^S_3(Z) = 0$. Therefore, by Definition 3.4, $[\![\varphi_1 \wedge \varphi_2]\!]^S_3(Z) = 0$.

Case 2: $[\![\varphi_1 \wedge \varphi_2]\!]^{S'}_3(f \; o \; Z) = 1$.
In this case, both $[\![\varphi_1]\!]^{S'}_3(f \; o \; Z) = 1$ and $[\![\varphi_2]\!]^{S'}_3(f \; o \; Z) = 1$. Then, by the induction hypothesis for φ_1 and φ_2, we conclude that $[\![\varphi_1]\!]^S_3(Z) = 1$ and
$[\![\varphi_2]\!]^S_3(Z) = 1$. Therefore, by Definition 3.4, $[\![\varphi_1 \wedge \varphi_2]\!]^S_3(Z) = 1$.

Logical-negation $\varphi = \neg\varphi_1$. The proof splits into the following subcases:

Case 1: $[\![\neg\varphi_1]\!]_3^{S'}(f\, o\, Z) = 0$.
In this case, $[\![\varphi_1]\!]_3^{S'}(f\, o\, Z) = 1$.
Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]_3^{S}(Z) = 1$.
Therefore, by Definition 3.4, $[\![\neg\varphi_1]\!]_3^{S}(Z) = 0$.

Case 2: $[\![\neg\varphi_1]\!]_3^{S'}(f\, o\, Z) = 1$.
In this case, $[\![\varphi_1]\!]_3^{S'}(f\, o\, Z) = 0$.
Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]_3^{S}(Z) = 0$.
Therefore, by Definition 3.4, $[\![\neg\varphi_1]\!]_3^{S}(Z) = 1$.

Existential-Quantification $\varphi = \exists v_0 : \varphi_1$. The proof splits into the following subcases:

Case 1: $[\![\exists v_1 : \varphi_1]\!]_3^{S'}(f\, o\, Z) = 0$.
In this case, for all $u \in U^S$, $[\![\varphi_1]\!]_3^{S'}(f\, o\, Z[v_1 \mapsto f(u)]) = 0$. Then, by the induction hypothesis for φ_1, we conclude that for all $u \in U^S$ $[\![\varphi_1]\!]_3^{S}(Z[v_1 \mapsto u]) = 0$. Therefore, by Definition 3.4, $[\![\exists v_1 : \varphi_1]\!]_3^{S}(Z) = 0$.

Case 2:$[\![\exists v_1 : \varphi_1]\!]_3^{S'}(f\, o\, Z) = 1$.
In this case, there exists a $u' \in U^{S'}$ such that $[\![\varphi_1]\!]_3^{S'}(f\, o\, Z[v_1 \mapsto u']) = 1$. Because f is surjective, there exists a $u \in U^S$ such that $f(u) = u'$ and $[\![\varphi_1]\!]_3^{S'}(f\, o\, Z[v_1 \mapsto f(u)]) = 1$. Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]_3^{S}(Z[v_1 \mapsto u]) = 1$. Therefore, by Definition 3.4, $[\![\exists v_1 : \varphi_1]\!]_3^{S}(Z) = 1$.

Transitive Closure $\varphi = (TC\ v_1, v_2 : \varphi_1)(v_3, v_4)$. The proof splits into the following subcases:

Case 1: $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S'}(f\, o\, Z) = 1$.
By Definition 3.4, there exist $u'_1, u'_2, \ldots, u'_{n+1} \in U^{S'}$ such that for all $1 \leq i \leq n$, $[\![\varphi_1]\!]_3^{S'}((f\, o\, Z)[v_1 \mapsto u'_i, v_2 \mapsto u'_{i+1}]) = 1$, $(f\, o\, Z)(v_3) = u'_1$, and $(f\, o\, Z)(v_4) = u'_2$. Because f is surjective, there exist $u_1, u_2, \ldots, u_{n+1} \in U^S$ such that for all $1 \leq i \leq n+1$, $f(u_i) = u'_i$. Therefore, $Z(v_3) = u_1$, $Z(v_4) = u_2$, and by the induction hypothesis, for all $1 \leq i \leq n$, $[\![\varphi_1]\!]_3^{S}(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}]) = 1$. Hence, by Definition 3.4, $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S}(Z) = 1$.

Case 2: $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S'}(f\, o\, Z) = 0$.
We need to show that $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S}(Z) = 0$. Assume on the contrary that $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S'}(f\, o\, Z) = 0$, but $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S}(Z) \neq 0$. Because $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S}(Z) \neq 0$, by Definition 3.4 there exist $u_1, u_2, \ldots, u_{n+1} \in U^S$ such that $Z(v_3) = u_1$, $Z(v_4) = u_2$, and for all $1 \leq i \leq n$, $[\![\varphi_1]\!]_3^{S}(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}]) \neq 0$. Hence, by the induction hypothesis there exist $u'_1, u'_2, \ldots, u'_{n+1} \in U^{S'}$ such that $(f\, o\, Z)(v_3) = u'_1$, and $(f\, o\, Z)(v_4) = u'_2$ and for all $1 \leq i \leq n$, $[\![\varphi_1]\!]_3^{S'}((f\, o\, Z)[v_1 \mapsto u'_i, v_2 \mapsto u'_{i+1}]) \neq 0$. Therefore, by Definition 3.4, $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S'}(f\, o\, Z) \neq 0$, which is a contradiction.

B A Soundness Proof for the Analysis

We extend operations on structures to operations on sets of structures in the natural way: For an operation *op* that returns a set (such as γ, $[\![st]\!]$, etc.),

$$\widehat{op}(XS) \stackrel{\text{def}}{=} \bigcup_{S \in XS} op(S). \tag{69}$$

For an operation *op* that returns either a single structure or $\perp$ (such as *coerce*),

$$\widehat{op}(XS) \stackrel{\text{def}}{=} \{S' \mid \exists S \in XS : S' = op(S) \wedge S' \neq \perp\}. \tag{70}$$

Showing that the shape-analysis algorithm is conservative amounts to showing that, for all statements *st*, the following holds:

$$\forall S \in \text{3-CSTRUCT}[\mathcal{P}, R(F)] : \widehat{[\![st]\!]}(\gamma(S)) \subseteq \widehat{\gamma}(\widehat{coerce}(\widehat{[\![st]\!]}(focus_{\{\varphi_x^{st} \mid \mathtt{x} \in PVar\}}(S)))) \tag{71}$$

However, it is easier to show the stronger claim that *focus* and *coerce* are semantic-reduction operations, i.e.,

$$\begin{array}{rcll} \forall S \in \text{3-STRUCT}[\mathcal{P}] : \gamma(S) & = & \widehat{\gamma}(\mathit{focus}_{\varphi}(S)) & (72)\\ \forall S \in \text{3-STRUCT}[\mathcal{P}] : \gamma(S) & = & \gamma(\mathit{coerce}(S)) & (73) \end{array}$$

These conditions are stronger than equation (71) because they imply that we can apply *focus* and *coerce* anywhere without losing precision. Moreover, equation (72) implies that focus_{φ} is sound for every formula φ, not just for the formulae φ_x^{st} used in Sections 5.1 and 5.1.2. Also, notice that when $\mathit{coerce}(S)$ returns $\perp$, by Definition 4.1 $\gamma(\mathit{coerce}(S))$ equals $\emptyset$, and therefore, in this case, equation (73) amounts to the statement that $\perp$ does not represent any concrete structure.

In fact, we will show something stronger than equation (72) by breaking *focus* down into smaller steps. The following operation is similar to *focus* except that the requirement of maximality is relaxed:

Definition B.1 *For a formula φ, the operation $\widetilde{\mathit{focus}}_{\varphi}$ yields the (possibly infinite) set of structures in which φ evaluates to a definite value, i.e.,*

$$\widetilde{\mathit{focus}}_{\varphi}(S) = \left\{ S' \;\middle|\; \begin{array}{l} S' \in \textit{3-STRUCT}[\mathcal{P}] \\ S' \sqsubseteq S \\ \forall Z, [\![\varphi]\!]_3^{S'}(Z) \neq 1/2 \end{array} \right\}$$

□

Obviously, for every structure S and formula φ, $\mathit{focus}_{\varphi}(S) = \mathit{maximal}(\widetilde{\mathit{focus}}_{\varphi}(S))$.

We now state the soundness lemma, which says that the operations *maximal*, *coerce*, and $\widetilde{\mathit{focus}}$ are indeed semantic reductions:

Lemma B.2 *For every structure $S \in$ 3-STRUCT$[\mathcal{P}]$, $XS \in 2^{\text{3-STRUCT}[\mathcal{P}]}$ and formula φ,*

$$\begin{array}{rcll} \widehat{\gamma}(\mathit{maximal}(XS)) & = & \widehat{\gamma}(XS) & (74)\\ \gamma(\mathit{coerce}(S)) & = & \gamma(S) & (75)\\ \widehat{\gamma}(\widetilde{\mathit{focus}}_{\varphi}(S)) & = & \gamma(S) & (76) \end{array}$$

Proof: Let us show that equation (74) holds. Clearly, $\subseteq$ holds. Therefore, let us show that $\supseteq$ holds. Let $S^{\natural} \in \widehat{\gamma}(XS)$, and thus by the definition of $\widehat{\gamma}$, there exists $S \in XS$ such that $S^{\natural} \in \gamma(S)$. Therefore, by the definition of γ, $S^{\natural} \sqsubseteq S$. Now for every S' such that $S \sqsubseteq S'$, we have $S^{\natural} \sqsubseteq S'$ by transitivity, and therefore $S^{\natural} \in \gamma(S')$. In particular this holds when S' is a maximal structure of XS.

Let us show that equation (75) holds. Clearly, $\subseteq$ holds. Therefore, let us show that $\supseteq$ holds. Let $S^{\natural} \in \gamma(S)$, and thus by Lemma 5.20, there exists $S' \in$ 3-CSTRUCT$[\mathcal{P}, R(F)]$ such that (i) $U^{S'} = U^S$, (ii) $S' \sqsubseteq S$ and (iii) $S^{\natural} \in \gamma(S')$. Therefore, by Definition 5.22, there exists $S'' \in \mathit{coerce}(S)$ such that $S' \sqsubseteq S''$. By the definition of γ, $S^{\natural} \in \gamma(S'')$

Let us show that equation (76) holds. Clearly, $\subseteq$ holds. Therefore, let us show that $\supseteq$ holds. Let $S^{\natural} \in \gamma(S)$, and thus by the definition of γ, $S^{\natural} \in$ 2-CSTRUCT$[\mathcal{P}, F]$ and $S^{\natural} \sqsubseteq S$. Now since $S^{\natural}$ is a two-valued structure, for every Z, we have $[\![\varphi]\!]_3^{S^{\natural}}(Z) \neq 1/2$, and thus $S^{\natural} \in \widetilde{\mathit{focus}}_{\varphi}(S)$. Since $\sqsubseteq$ is reflexive, $S^{\natural} \in \widehat{\gamma}(\widetilde{\mathit{focus}}_{\varphi}(S))$. □

References

[AW93] 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.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *ACM Symposium on Principles of Programming Languages*, pages 269–282, New York, NY, 1979. ACM Press.

[CWZ90] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 296–310, New York, NY, 1990. ACM Press.

[Deu92] 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.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 230–241, New York, NY, 1994. ACM Press.

[Eva96] D. Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Languages Design and Implementation*, 1996. Available at "http://larch-www.lcs.mit.edu:8001/~ evs/pldi96-abstract.html".

[Gin88] M.L. Ginsberg. Multivalued logics: A uniform approach to inference in artificial intelligence. *Computer Intelligence*, 4:265–316, 1988.

[Hen90] L. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, Ithaca, N.Y., Jan 1990.

[HHN92] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 249–260, New York, NY, June 1992. ACM Press.

[HN90] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[Hoa75] C.A.R. Hoare. Recursive data structures. *International Journal of Computer and Information Sciences*, 4(2):105–132, 1975.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 28–40, New York, NY, 1989. ACM Press.

[JM81] 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.

[JM82] N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *ACM Symposium on Principles of Programming Languages*, pages 66–74, New York, NY, 1982. ACM Press.

[Kle87] S.C. Kleene. *Introduction to Metamathematics.* North-Holland, second edition, 1987.

[LH88] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 21–34, New York, NY, 1988. ACM Press.

[LM96] C.-K. Luk and T.C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[LR91] W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages*, pages 93–103, New York, NY, January 1991. ACM Press.

[PCK93] 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 *Lecture Notes in Computer Science*, pages 37–57, Portland, OR, August 1993. Springer-Verlag.

[SRW96] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *ACM Symposium on Principles of Programming Languages*, New York, NY, January 1996. ACM Press.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[Str92] J. Stransky. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Information and Computation*, 101(1):70–102, November 1992.

[Wan94] E. Y.-B. Wang. *Analysis of Recursive Types in an Imperative Language.* PhD thesis, University of California, Berkeley, CA, 1994.