Parametric Shape Analysis via 3-Valued Logic

MOOLY SAGIV
and
THOMAS REPS
and
REINHARD WILHELM

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. A key innovation of this work is that the stores that can possibly arise during execution are represented using 3-valued logical structures.

Questions about properties of stores can be answered by evaluating predicate-logic formulae using Kleene's semantics of 3-valued logic:

—If a formula evaluates to *true*, then the formula holds in every store represented by the 3-valued structure.
—If a formula evaluates to *false*, then the formula does not hold in any store represented by the 3-valued structure.
—If a formula evaluates to *unknown*, then we do not know if this formula always holds, never holds, or sometimes holds and sometimes does not hold in the stores represented by the 3-valued structure.

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

This paper presents a *parametric* framework for shape analysis: It provides the basis for generating different shape-analysis algorithms by varying the predicates used in the 3-valued logic. The analysis algorithms generated handle every program, but may produce conservative results due to the class of predicates employed; that is, different 3-valued logics may be needed, depending on the kinds of linked data structures used in the program and on the link-rearrangement operations performed by the program's statements.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—

A preliminary version of this paper appeared in the Proc. of the 1999 ACM Symp. on Princ. of Prog. Lang. [Sagiv et al. 1999]. Part of this research was done while Sagiv was at the University of Chicago. Sagiv was supported in part by the National Science Foundation under grant CCR-9619219 and 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.
Reps was 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, 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.
Wilhelm was 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.

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 and Phrases: Abstract interpretation, alias analysis, constraint solving, dataflow analysis, destructive updating, pointer analysis, shape analysis, static analysis, 3-valued logic

1. INTRODUCTION

Data structures built using pointers can be characterized by invariants describing their “shape” at stable states, i.e., in between operations on them. These invariants are usually not preserved by the execution of individual program statements, and it is challenging to prove that invariants are reestablished once a sequence of operations is finished [Hoare 1975]. Such invariants are useful for sharpening the results obtained from a tool like `LClint`, which predicts memory-usage bugs [Evans 1996], and for program optimization (e.g., to improve memory locality [Luk and Mowry 1996]).

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 [Jones and Muchnick 1981; 1982; Larus and Hilfinger 1988; Horwitz et al. 1989; Chase et al. 1990; Stransky 1992; Assmann and Weinhardt 1993; Plevyak et al. 1993; Wang 1994; Sagiv et al. 1998]. A common feature of these algorithms is that they represent heap cells by “shape-nodes” and sets of “indistinguishable” heap cells by a single shape-node, often called a *summary-node* [Chase et al. 1990]. In these shape analyses, the shape graphs capture properties of the stores that arise at the different points in the program.

1.1 Main Results

1.1.1 *Parametricity.* This paper presents a *parametric* framework for shape analysis. Different instantiations of the framework create analyses that use different classes of shape graphs, and hence are prepared to identify different classes of store properties that hold at the different points in a program. The analysis algorithms handle every program, but may produce conservative results due to the use of an inappropriate class of shape graphs; that is, different classes of shape graphs may be needed, depending on the kinds of linked data structures used in a program and on the link-rearrangement operations performed by the program's statements.

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. A prototype version of such a system, based on the methods presented in this paper, has recently been implemented in Java [Lev-Ami 2000].

A number of previous shape-analysis algorithms, including [Jones and Muchnick 1981; 1982; Horwitz et al. 1989; Chase et al. 1990; Stransky 1992; Plevyak et al. 1993; Wang 1994; Sagiv et al. 1998], can be viewed as instances of the framework presented in this paper.

1.1.2 *The Use of Logic for Shape Analysis.* In our shape-analysis framework, predicate-logic formulae play many roles: expressing both the concrete and abstract semantics of the programming language; expressing properties of store elements (e.g., may-aliases, must-aliases); and expressing properties of stores (e.g., data-structure invariants). For instance, the formula $x(v)$ expresses whether pointer variable `x` points to heap cell v; the formula $n(v_1, v_2)$ express whether the `n`-field of heap cell v_1 points to heap cell v_2; to express the property "program variables `x` and `y` are not may-aliases", we write the formula

$$\forall v : \neg(x(v) \wedge y(v)); \qquad (1)$$

to specify the effect of the execution of the statement "`x = x->n`" on variable `x` (part of the concrete semantics), we write the formula

$$x'(v) \stackrel{\text{def}}{=} \exists v_1 : x(v_1) \wedge n(v_1, v). \qquad (2)$$

Formula (2) indicates that after the statement `x = x->n`, variable `x` points to a heap cell that was formerly pointed to by `x->n`.

1.1.3 *Shape Analysis via* 3*-Valued Logic.* We use Kleene's 3-valued logic [Kleene 1987] to create a shape-analysis algorithm automatically from a specification. Kleene's logic, which has a third truth value that signifies "unknown", 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 3-valued logic is that the interpretations of formulae in 2-valued and 3-valued logic coincide on *true* and *false*. This comes in handy for shape analysis, where we wish to relate the concrete (2-valued) world and the abstract (3-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 a statement about the 2-valued world or the 3-valued world.

In this paper, shape graphs are represented as "3-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 (1) 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 (1) 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 tell us 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 [Jones and Muchnick 1981; Horwitz et al. 1989; Chase et al. 1990; Stransky 1992].

1.1.4 *Materialization of New Nodes from Summary Nodes.* One of the magical aspects of [Sagiv et al. 1998] is "materialization", in which the transfer function that expresses the semantics of statements of the form `y = x->n` can split a summary-node into two separate nodes. (This operation is also discussed in [Chase et al. 1990; Plevyak et al. 1993].) 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 [Sagiv et al. 1998], it was observed that node materialization is complicated because various kinds of shape-graph properties are interdependent. 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 (3-valued) constraints on shape graphs, which we use to capture such dependences between properties. In Section 5.2.3, we give an algorithm that solves systems of such constraints.

1.1.5 *Separation of Disjoint Structures.* The framework allows us to create algorithms that are more precise than the above-cited shape-analysis algorithms. In particular, by tracking which heap cells are *reachable* from which program variables, it is often possible to determine precise shape information for programs that manipulate several (possibly cyclic) data structures (see Section 6.1). Other static-analysis techniques (including ones that are not based on shape graphs [Landi and Ryder 1991; Hendren 1990; Hendren and Nicolau 1990; Deutsch 1992; 1994]) yield very imprecise information on these programs.

1.1.6 *Relating* 2*-Valued Logic to* 3*-Valued Logic.* The fundamental logical principles on which our shape-analysis framework relies appear to be new. To relate 2-valued logic to 3-valued logic, we introduce a general notion of "truth-blurring" embeddings that map from a 2-valued world to a corresponding 3-valued one. Our Embedding Theorem (Theorem 3.11) ensures that the meaning of a formula in the "blurred" (3-valued) world is consistent with the formula's meaning in the original (2-valued) world.

1.2 Limitations

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

—There are other shape-analysis algorithms—not based on shape graphs—that are incomparable to our method. For example, the algorithm of [Hendren 1990;

Hendren and Nicolau 1990] handles destructive updates in many cases, but does not handle cyclic or doubly-linked lists. The algorithm of [Deutsch 1992; 1994] 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 [Chase et al. 1990; Assmann and Weinhardt 1993; Sagiv et al. 1998]. Because the intraprocedural versions of these algorithms are instances of our framework, their methods for handling procedures should generalize to the parametric case.

—The number of possible shape-nodes that may arise during abstract interpretation is potentially exponential in the size of the specification. We do not know how severe this problem is in practice. However, it is possible to define a widening operator that converts a shape graph into a more compact, but possibly less precise, shape graph by collapsing more nodes into summary nodes. This can be used to make a shape-analysis algorithm polynomial, at the cost of making the results less accurate.

—The number of shape graphs may be quite large (as in [Jones and Muchnick 1981; Horwitz et al. 1989]). This problem was avoided in [Larus and Hilfinger 1988; Chase et al. 1990; Plevyak et al. 1993; Sagiv et al. 1998] 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 Prototype Implementation

The algorithms presented in this paper have been implemented by T. Lev-Ami (see [Lev-Ami 2000]). This implementation has been used to test our ideas and has led to a number of improvements in them.

1.4 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 3-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 instantiations of the parametric framework. Section 7 discusses related work. Section 8 consists of some final remarks. The proof of the main logical theorem on which our approach relies and other technical proofs are presented in Appendices A and B.

2. AN OVERVIEW OF THE PARAMETRIC FRAMEWORK

Fig. 1(a) shows the declaration of a linked-list data type in C, and Fig. 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 many examples given in the remainder of the paper. The `reverse` program allows us to demonstrate many aspects of the shape-analysis

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

(a)

```
/* 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)

Fig. 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.

framework in a nontrivial, but still relatively digestible, fashion.

2.1 Representing Stores via 3-Valued Structures

In Section 1, we couched the discussion in terms of shape-graphs for the convenience of readers who are familiar with previous work. Formally, we do not work with shape-graphs; instead, the abstractions of stores will be what logicians call 3-*valued logical structures*, denoted by $\langle U, \iota \rangle$. There is a *vocabulary* of predicate symbols (with given arities); each logical structure has a universe of *individuals* U, and ι maps each possible tuple $p(u_1, \ldots, u_k)$ of an arity-k predicate symbol p, where $u_i \in U$, to the value 0, 1, or 1/2, (i.e., *false*, *true*, and *unknown*, respectively). Logical structures are used to encode stores as follows: Individuals represent abstractions of memory locations in the heap; 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 3-valued structures that describe all possible inputs to `reverse` are shown in the second column of Fig. 2. The following graphical notation is used for 3-valued logical structures (cf. column 3 of Fig. 2):

—Individuals of the universe are represented by circles with names inside.

—Summary nodes (i.e., those for which $sm = 1/2$) are represented by dotted circles.

—Other unary predicates with value 1 (1/2) and binary pointer-component-points-to predicates are represented by solid (dotted) arrows.

Thus, in structure S_2, pointer variable x points to element u_1, whose n field may point to a location represented by element u. u is a summary node, i.e., it may represent more than one location. Possibly there is an n field in one of these locations that points to another location represented by u.

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

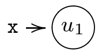
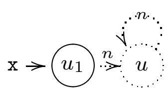
<table>
<tr><th>S</th><th>Logical Structure</th><th>Graphical Representation</th></tr>
<tr><td>S_0</td><td>unary predicates:
<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>sm</th><th>is</th></tr></table>
binary predicates:
<table><tr><th>n</th></tr></table></td><td></td></tr>
<tr><td>S_1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

Fig. 2. The 3-valued logical structures that describe all possible acyclic inputs to `reverse`.

—The abstract element u_1 represents the head of the list, and u represents all of 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, $y(u) = 0$, $y(u_1) = 0$, $t(u) = 0$, and $t(u_1) = 0$ 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.

—The binary predicate n represents the `n` fields of list elements. The value of $n(u_1, u)$ is $1/2$ because there is a list element represented by u that is the immediate `n`-successor of u_1, but other list elements represented by u are not the immediate `n`-successor of u_1.

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

The 3-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.

$\wedge$	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 I. Kleene's 3-valued interpretation of the propositional operators.

—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

3-valued structures offer a systematic way to answer questions about properties of the stores they represent:

OBSERVATION 2.1. **[Property-Extraction Principle]**. *Questions about properties of stores can be answered by evaluating formulae using Kleene's semantics of* 3*-valued logic:*

—If a formula evaluates to 1*, then the formula holds in every store represented by the* 3*-valued structure.*

—If a formula evaluates to 0*, then the formula never holds in any store represented by the* 3*-valued structure.*

—If a formula evaluates to 1/2*, then we do not know if this formula always holds, never holds, or sometimes holds and sometimes does not hold in the stores represented by the* 3*-valued structure.*

Kleene's 3-valued interpretation of the propositional operators is given in Table I.

In Section 3.4, we give the *Embedding Theorem* (Theorem 3.11), which states that the 3-valued Kleene interpretation in S of every formula is consistent with the formula's 2-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

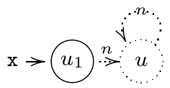
$$\exists v : x(v) \wedge n(v, v),$$

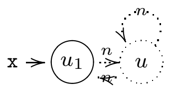
which expresses the property "`x` points to a cell that has a self-cycle", evaluates to 0 because `x` and `x->n` point to different elements in all of the stores represented by S_2. This information can be used by an optimizing compiler to determine whether it is profitable to generate a prefetch for the next element [Luk and Mowry 1996].

Now consider the formula

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

which expresses the property "Do two or more different heap cells point to heap cell v?" Formula $\varphi_{is,n}(v)$ 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 heap cells point to u.

<table>
<tr><th></th><th>Logical Structure</th><th>Graphical Representation</th></tr>
<tr><td>Acyclic List</td><td>
unary predicates:
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>sm</th><th>is</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>0</td><td>0</td><td>1/2</td><td>0</td></tr>
</table>
binary predicates:
<table>
<tr><th>n</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>0</td><td>1/2</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>
</td><td>


</td></tr>
<tr><td>Possibly Cyclic List</td><td>
unary predicates:
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>sm</th><th>is</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>0</td><td>0</td><td>1/2</td><td>0</td></tr>
</table>
binary predicates:
<table>
<tr><th>n</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>0</td><td>1/2</td></tr>
<tr><td>u</td><td>1/2</td><td>1/2</td></tr>
</table>
</td><td>


</td></tr>
</table>

Fig. 3. The shape graphs for acyclic and possibly cyclic lists.

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 3-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, instrumentation predicates play a key role in the *parametric framework* for shape analysis based on abstract interpretation. In general, adding additional instrumentation predicates refines the abstraction used for shape analysis; 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.

Fig. 3 demonstrates how acyclic and possibly cyclic lists are represented by different shape graphs. In the shape graph that represents possibly cyclic lists, the backpointer back to the head of the list is represented by the fact that the value of $n(u, u_1)$ is 1/2. Note that the value of $is(u)$ is still 0.

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 evaluating a suitable collection of formulae that capture the semantics of st.*

Evaluation of the formulae in 2-valued logic captures the transfer function for st of the concrete semantics. Evaluation of the formulae in 3-valued logic captures the transfer function for st of the abstract semantics.

Observation 2.3 allows us to simplify drastically the argument that the shape-analysis framework is correct (compared, for example, to our previous work [Sagiv et al. 1998]), because the correctness of the abstract semantics falls out directly from the Embedding Theorem (Theorem 3.11).

Fig. 4 illustrates the first two iterations of an abstract interpretation of `reverse` on the structure S_2 from Fig. 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 Fig. 4. To simplify the presentation, in Fig. 4 (and elsewhere) we break each occurrence of st_5: `y->n = t` into two statements: $st_{5.1}$: `y->n = NULL`, followed by $st_{5.2}$: `y->n = t`, so that in the predicate-update formulae for $st_{5.2}$ we can assume that `y->n == NULL`. Fig. 4 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.

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 3-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 Fig. 4 is essentially that of Chase et al. [Chase et al. 1990]. Others that are amenable to being simulated in this fashion include [Jones and Muchnick 1981; Larus and Hilfinger 1988; Horwitz et al. 1989].

Unfortunately, there is also bad news: The method described above and illustrated in Fig. 4 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).

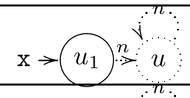
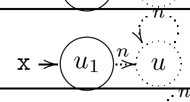
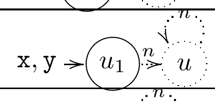
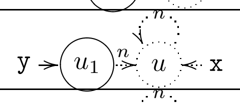
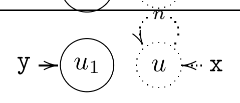
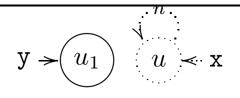
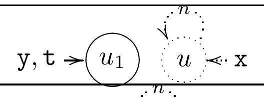
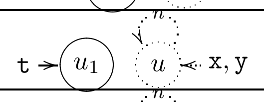
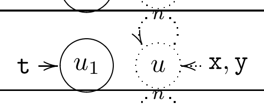
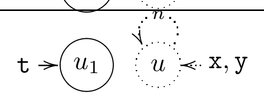
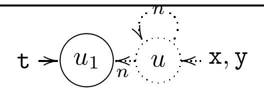
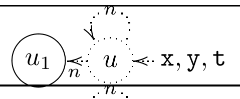
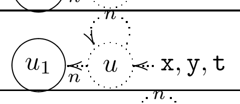
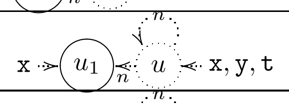
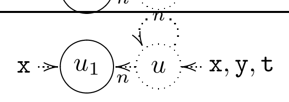
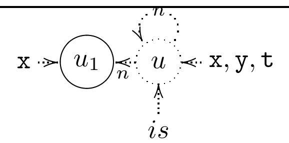
Statement	Formula	Structure After
st_1: `y = NULL;`	$y'(v) = \mathbf{0}$	 S_3
st_2: `t = y;`	$t'(v) = y(v)$	 S_4
st_3: `y = x;`	$y'(v) = x(v)$	 S_5
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$	 S_6
$st_{5.1}$: `y->n = NULL;`	$n'(v_1, v_2) = n(v_1, v_2) \wedge \neg y(v_1)$ $is'(v) = \begin{cases} is(v) \wedge \varphi_{is,n'} & \text{if } \exists v' : y(v') \wedge n(v', v) \\ is(v) & \text{otherwise} \end{cases}$	 S_7
$st_{5.2}$: `y->n = t;`	$n'(v_1, v_2) = n(v_1, v_2) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = \begin{cases} is(v) \vee \varphi_{is,n'} & \text{if } \exists v_1 : t(v) \wedge n(v_1, v) \\ is(v) & \text{otherwise} \end{cases}$	 S_8
st_2: `t = y;`	$t'(v) = y(v)$	 S_9
st_3: `y = x;`	$y'(v) = x(v)$	 S_{10}
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$	 S_{11}
$st_{5.1}$: `y->n = NULL;`	$n'(v_1, v_2) = n(v_1, v_2) \wedge \neg y(v_1)$ $is'(v) = \begin{cases} is(v) \wedge \varphi_{is,n'} & \text{if } \exists v' : y(v') \wedge n(v', v) \\ is(v) & \text{otherwise} \end{cases}$	 S_{12}
$st_{5.2}$: `y->n = t;`	$n'(v_1, v_2) = n(v_1, v_2) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = \begin{cases} is(v) \vee \varphi_{is,n'} & \text{if } \exists v_1 : t(v) \wedge n(v_1, v) \\ is(v) & \text{otherwise} \end{cases}$	 S_{13}
st_2: `t = y;`	$t'(v) = y(v)$	 S_{14}
st_3: `y = x;`	$y'(v) = x(v)$	 S_{15}
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$	 S_{16}
$st_{5.1}$: `y->n = NULL;`	$n'(v_1, v_2) = n(v_1, v_2) \wedge \neg y(v_1)$ $is'(v) = \begin{cases} is(v) \wedge \varphi_{is,n'} & \text{if } \exists v' : y(v') \wedge n(v', v) \\ is(v) & \text{otherwise} \end{cases}$	 S_{17}
$st_{5.2}$: `y->n = t;`	$n'(v_1, v_2) = n(v_1, v_2) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = \begin{cases} is(v) \vee \varphi_{is,n'} & \text{if } \exists v_1 : t(v) \wedge n(v_1, v) \\ is(v) & \text{otherwise} \end{cases}$	 S_{18}

Fig. 4. The first three iterations of the simple abstract interpretation of `reverse` applied to structure S_2 shown in Fig. 2 (which represents acyclic lists of length two or more).

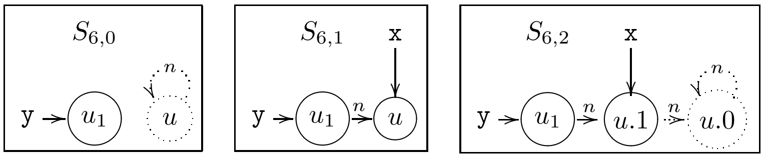


Fig. 5. The three structures that result from the first abstract execution of st_4 by the improved abstract-interpretation method of Section 5.

—This eventually leads to the situation that occurs after the third abstract execution of st_4, which produces structure S_{16}. Structure S_{16} 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 traversed. (Thus, Section 5 generalizes the algorithm of [Sagiv et al. 1998].) As we will see in Section 5, this allows us to determine the correct shape invariants 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. [Plevyak et al. 1993] introduced a way to do materialization for straight-line code, and Sagiv et al. [Sagiv et al. 1998] 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 present a systematic solution to the materialization problem that is relatively easy to understand and prove correct. It is based on the following principle:

Observation 2.4. **[Materialization Principle]**. *Materialization is driven by a mechanism that refines a 3-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, Fig. 5 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 Fig. 5, $x(v)$ evaluates to 0 or 1, and not 1/2.

3. 3-VALUED LOGIC AND EMBEDDING

This section defines a 3-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:

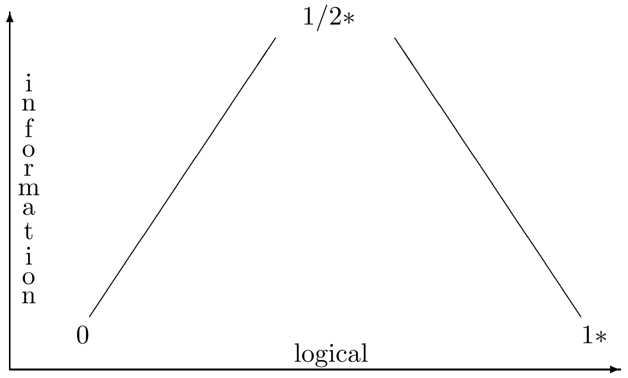


Fig. 6. The semi-bilattice of 3-valued logic. (The $*$ symbols attached to 1/2 and 1 indicate that these are the "designated values", which indicate "potential truth".)

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 $l_2 = 1/2$. The symbol $\sqcup$ denotes the least-upper bound operation with respect to $\sqsubseteq$.*

Kleene's semantics of 3-valued logic is monotonic in the information order (see Table I and Definition 3.4).

The values 0, 1, and 1/2 form a mathematical structure known as a semi-bilattice, e.g., [Ginsberg 1988], as shown in Fig. 6. A semi-bilattice has two orderings: the *logical order* and the *information order*. The logical order is the one used in Table I: 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 Fig. 6, a value that is "far enough to the right" in the logical order indicates "potential truth" (and is called a *designated value*). In the semi-bilattice of Fig. 6 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).

3.1 Syntax of First-Order Formulae with Transitive Closure

Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be a finite set of predicate symbols. Without loss of generality we exclude constant and function symbols from our logic.[1] We write first-order formulae over $\mathcal{P}$ using the logical connectives $\wedge$, $\vee$, $\neg$, and the quantifiers $\forall$ and $\exists$. The symbol $=$ denotes the equality predicate. The operator 'TC' denotes transitive closure on formulae. We also use several shorthand notations: for a binary predicate p, $p^+(v_3, v_4)$ is a shorthand for $(TC\ v_1, v_2 : p(v_1, v_2))(v_3, v_4)$; $\varphi_1 \Rightarrow \varphi_2$ is a shorthand for $(\neg\varphi_1 \vee \varphi_2)$; $\varphi_1 \Leftrightarrow \varphi_2$ is a shorthand for $(\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$, and $v_1 \neq v_2$ is a shorthand for $\neg(v_1 = v_2)$. Finally, we make use of conditional

[1]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 II. The core predicates that correspond to the `List` data-type declaration from Fig. 1(a).

expressions:

$$\begin{cases} \varphi_2 & \text{if } \varphi_1 \\ \varphi_3 & \text{otherwise} \end{cases} \quad \text{is a shorthand for } (\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3).$$

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

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

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

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

The formula $(v_1 = v_2)$, *where* v_1 *and* v_2 *are distinct variables, is an atomic formula with free variables* $\{v_1, v_2\}$.

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* $v_3, v_4 \notin V$, *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.*

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

Example 3.3. Table II contains the core-predicates for the `List` data-type declaration from Fig. 1(a) and the `reverse` program of Fig. 1(b). The unary predicate $sm \in \mathcal{C}$ captures the essence of "summary-nodes", which were introduced by Jones and Muchnick [Jones and Muchnick 1981] 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

Pred.	Intended Meaning	Purpose	Ref.
$is(v)$	Do two or more fields of heap elements point to v?	lists and trees	[Chase et al. 1990], [Sagiv et al. 1998]
$r_x(v)$	Is v (transitively) reachable from pointer variable `x`?	separating disjoint data structures	[Sagiv et al. 1998]
$r(v)$	Is v reachable from some pointer variable (i.e., is v a non-garbage element)?	compile-time garbage collection	
$c(v)$	Is v on a directed cycle?	reference counting	[Jones and Muchnick 1981]
$c_{f.b}(v)$	Does a field-`f` deref. from v, followed by a field-`b` deref., yield v?	doubly-linked lists	[Hendren et al. 1992], [Plevyak et al. 1993]
$c_{b.f}(v)$	Does a field-`b` deref. from v, followed by a field-`f` deref., yield v?	doubly-linked lists	[Hendren et al. 1992], [Plevyak et al. 1993]

Table III. Examples of instrumentation predicates.

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

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

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

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

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

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

Table IV. Formulae for the instrumentation predicates listed in Table III.

example, in structure S_2 from Fig. 2, u_1 represents a unique concrete element of any store that 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. However, 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 Fig. 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). Notice that in this structure $n(u, u)$ cannot have the value 1 because u represents a unique, non-shared heap cell (in particular, $is(u) = 0$). Therefore, the structure S_2' and the structure S_2'' in which $n(u, u) = 0$ represent the same set of concrete stores.

Table III lists some interesting instrumentation predicates, and Table IV lists their defining formulae.

—The sharing predicate is was introduced in [Chase et al. 1990] and also used in [Sagiv et al. 1998] to capture list and tree data structures.

—The reachable-from-variable-x predicate r_x was mentioned in [Sagiv et al. 1998, p.38]. It serves to differentiate different summary nodes, and thus separates the abstract representations of data structures that are disjoint in the concrete world. This leads to increased precision in many programs, including programs that manipulate singly linked lists. (See Section 6.1.1.)

—The reachability predicate r identifies non-garbage cells. This is useful for determining when compile-time garbage collection can be performed. (See Section 6.1.2.)

—The cyclicity predicate c was introduced by Jones and Muchnick [Jones and Muchnick 1981] to aid in determining when reference counting would be sufficient. (See Section 6.1.1.)

—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 [Hendren et al. 1992] and also used in [Plevyak et al. 1993]. (See Section 6.2.)

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

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

3.2 Kleene's 3-Valued Semantics

In this section, we define Kleene's 3-valued semantics for first-order formulae with transitive closure.

Definition 3.4. *A* **3-valued interpretation** *of the language of formulae over* $\mathcal{P}$ *is a* **3-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:*

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

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 $\boldsymbol{l} \in \{\mathbf{0}, \mathbf{1}, \mathbf{1/2}\}$, $[\![\boldsymbol{l}]\!]_3^S(Z) = l$ *(where* $l \in \{0, 1, 1/2\}$*).*

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

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

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

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

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 $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$,

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

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 Fig. 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 heap cells point to heap cell v?". For the assignment $Z_1 = [v \mapsto u]$, we have

$$\begin{aligned} [\![\varphi_{is}]\!]_3^S(Z_1) &= \max_{u', u'' \in \{u_1, u\}} [\![n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2]\!]_3^S([v \mapsto u, v_1 \mapsto u', v_2 \mapsto u'']) \\ &= 1/2, \end{aligned}$$

and thus $S_2, Z_1 \models \varphi_{is}$. In contrast, for the assignment $Z_2 = [v \mapsto u_1]$, we have

$$\begin{aligned} [\![\varphi_{is}]\!]_3^S(Z_2) &= \max_{u', u'' \in \{u_1, u\}} [\![n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2]\!]_3^S([v \mapsto u_1, v_1 \mapsto u', v_2 \mapsto u'']) \\ &= 0, \end{aligned}$$

and thus $S_2, Z_2 \not\models \varphi_{is}$.

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 '='):[2]

[2]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.

—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.

Notice that Definition 3.4 could be generalized to allow many-sorted sets of individuals. This would be useful for modeling heap cells of different types; however, to simplify the presentation, we have chosen not to introduce this mechanism.

3.3 Properties of 3-Valued Logic

3-valued logic retains a number of properties that are familiar from 2-valued logic:

LEMMA 3.6. *Let φ_1, φ_2, and φ_3 be formulae, let S be a 3-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.

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

Associativity Laws.

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

Commutativity Laws.

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

Distributivity Laws.

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

Implication Law.

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

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, $\iota^S(p)(u_1, \ldots, u_k) \sqsubseteq \iota^{S'}(p)(u_1, \ldots, u_k)$.) 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 2-valued and 3-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 $u_1, \ldots, u_k \in U^S$,*

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

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, and therefore $\iota^{S'}(sm)(u')$ can never be 1.

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,*

$$\iota^{S'}(p)(u'_1, \ldots, u'_k) = \bigsqcup_{t\_embed(u_i) = u'_i, 1 \le i \le k} \iota^S(p)(u_1, \ldots, u_k) \tag{26}$$

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., $S \sqsubseteq^{t\text{-}embed} t\_embed(S)$.

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!

Equation (27) has the form that it does so that tight embeddings compose properly (i.e., so that $t\_embed_2(t\_embed_1(S)) = (t\_embed_2 \circ t\_embed_1)(S)$ holds).

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

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

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

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 let $f : U^S \to U^{S'}$ be a function such that $S \sqsubseteq^f S'$. Then, for every formula φ and complete assignment Z for φ, $[\![\varphi]\!]_3^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S'}(f \circ Z)$.*
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:

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

$$
\begin{aligned}
1 &= [\![v = v]\!]_3^S([v \mapsto u_1]) \\
&\sqsubseteq [\![v = v]\!]_3^{S'}(t\_embed \circ [v \mapsto u_1]) \\
&= [\![v = v]\!]_3^{S'}([v \mapsto t\_embed(u_1)]) \\
&= [\![v = v]\!]_3^{S'}([v \mapsto u']) \\
&= 1/2
\end{aligned}
$$

The Embedding Theorem requires that f be surjective in order to guarantee that a quantified formula, such as $\exists v : \varphi$, has consistent values in S and S'. For example, 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 $[\![\exists v : \varphi]\!]_3^{S'}(f \circ Z) = 1$.

Apart from surjectivity, the Embedding Theorem depends on the fact that the 3-valued meaning function is monotonic in its "interpretation" argument (cf. Lemma 3.7).

As mentioned in the Introduction, one of the nice properties of Kleene's 3-valued logic is that it coincides with 2-valued logic on the two values 0 and 1. This is useful for shape analysis, because we wish to relate concrete (2-valued) structures and abstract (3-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 2-valued structure or a 3-valued structure. The Embedding Theorem (Theorem 3.11) gives us the tool to relate the 2-valued and 3-valued interpretations.

3.5 Compatible Structures

The 2-valued logic that we have defined is slightly nonstandard in that (i) we assume that the core predicate sm is always present in $\mathcal{P}$, and (ii) the semantics of

$(v_1 = v_2)$ is defined in terms of $\iota(sm)$. The motivation for this is that sm is useful for defining the link between 2-valued and 3-valued logic.

We use 3-STRUCT[$\mathcal{P}$] to denote the set of general 3-valued structures over vocabulary $\mathcal{P}$, and 2-STRUCT[$\mathcal{P}$] to denote the set of 2-valued structures over $\mathcal{P}$, where in both cases we impose the restriction that for all u, $\iota^S(sm)(u) \neq 1$:

—For structures in 2-STRUCT[$\mathcal{P}$], the reason for the restriction that for all u, $\iota^S(sm)(u) = 0$ is to make the interpretation of $=$ coincide with the identity relation on individuals—and to avoid letting 1/2 creep into the semantics of formulae. For example, suppose that S were a structure in 2-STRUCT[$\mathcal{P}$] in which $\iota^S(sm)(u) = 1$: Under these circumstances, $[\![v_1 = v_2]\!]_3^S([v_1 \mapsto u, v_2 \mapsto u]) = 1/2$; that is, the meaning of an atomic formula with respect to a 2-valued interpretation can be 1/2. Consequently, to capture conventional 2-valued logic, we are interested only in 2-valued structures in which for all u, $\iota^S(sm)(u) = 0$. Alternatively, we say that we are interested only in 2-valued structures in which the *compatibility formula* $\forall v : \neg sm(v)$ is satisfied.

—For structures in 3-STRUCT[$\mathcal{P}$], the restriction that for all u, $\iota^S(sm)(u) \neq 1$ is a consequence of Definition 3.8.

Note that 2-STRUCT[$\mathcal{P}$] $\subseteq$ 3-STRUCT[$\mathcal{P}$].

We have other uses for the notion of compatibility formulae. For instance, suppose that P is a C program that operates on the `List` data-type of Fig. 1(a), and that $S^\natural \in$ 2-STRUCT[$\mathcal{P}$] is a 2-valued structure over the appropriate vocabulary. As described in Table II, 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 III.) 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. Again, we are not interested in all structures in 2-STRUCT[$\mathcal{P}$], but only in ones compatible with the semantics of C. Table V 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 Fig. 1(a). Formula (28) captures the condition that all sm predicate values are 0 in concrete stores. Formula (29) captures the fact that every program variable points to at most one list element. Formula (30) captures a similar property of the `n` fields of `List` structures: Whenever the `n` field of a list element is non-`NULL`, it points to at most one list element.

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

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

$$\forall v : \neg sm(v) \tag{28}$$
$$\text{for each } \mathtt{x} \in PVar, \forall v_1, v_2 : x(v_1) \wedge x(v_2) \Rightarrow v_1 = v_2 \tag{29}$$
$$\forall v_1, v_2 : (\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2)) \Rightarrow v_1 = v_2 \tag{30}$$
$$\forall v : (\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \Rightarrow is(v) \tag{31}$$
$$\forall v : \neg(\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \Rightarrow \neg is(v) \tag{32}$$

$$\text{for each } \mathtt{x} \in PVar, \forall v_2 : (\exists v_1 : x(v_1) \wedge v_1 \neq v_2) \Rightarrow \neg x(v_2) \tag{33}$$
$$\text{for each } \mathtt{x} \in PVar, \forall v_1 : (\exists v_2 : x(v_2) \wedge v_1 \neq v_2) \Rightarrow \neg x(v_1) \tag{34}$$
$$\forall v_2, v_3 : (\exists v_1 : n(v_3, v_1) \wedge v_1 \neq v_2) \Rightarrow \neg n(v_3, v_2) \tag{35}$$
$$\forall v_1, v_3 : (\exists v_2 : n(v_3, v_2) \wedge v_1 \neq v_2) \Rightarrow \neg n(v_3, v_1) \tag{36}$$
$$\forall v_2, v : (\exists v_1 : \neg is(v) \wedge n(v_1, v) \wedge v_1 \neq v_2) \Rightarrow \neg n(v_2, v) \tag{37}$$
$$\forall v_1, v : (\exists v_2 : \neg is(v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \Rightarrow \neg n(v_1, v) \tag{38}$$
$$\forall v_1, v_2 : (\exists v : \neg is(v) \wedge n(v_1, v) \wedge n(v_2, v)) \Rightarrow v_1 = v_2 \tag{39}$$

Table V. Compatibility formulae F for structures that represent a store of the `reverse` program, which operates on the `List` data-type declaration from Fig. 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.

This is then broken into two formulae of the form:

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

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

The rules below the line in Table V are logical consequences of the rules above the line, and are generated systematically from them, as explained in Section 5.2.1.

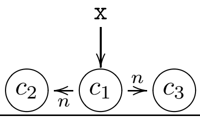
In the remainder of the paper, 2-CSTRUCT$[\mathcal{P}, F]$ denotes the set of 2-valued structures that satisfy a set of compatibility formulae F.

We can exploit the close relationship between 2-valued and 3-valued logic to extend the hygiene conditions to 3-valued structures. As with the 2-valued structures 2-STRUCT$[\mathcal{P}]$, the set of 3-valued structures 3-STRUCT$[\mathcal{P}]$ is more general than is necessary for shape analysis. One way to impose hygiene conditions on 3-valued structures is merely to use the same set of compatibility formulae F that we use for 2-valued structures, but to interpret the formulae in F under the 3-valued interpretation (i.e., Definition 3.4). By the Embedding Theorem, this is safe: Because we are only concerned with 2-valued structures $S^\natural \in$ 2-STRUCT$[\mathcal{P}]$ that *satisfy* all of the formulae in F, we need only be interested in 3-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 3-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 3-valued structure. In Section 4.2, the meaning functions of the program statements



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

Fig. 7. This structure, $S^\natural_{weird}$, is not represented by the structure S_2 from Fig. 2.

and conditions are defined. In Section 4.3, we address the question of defining appropriate formulae for updating instrumentation predicates.

To guarantee that the analysis of a program containing a loop terminates, we require that the number of potential structures for a given program be finite. For this reason, in Section 4.4 we introduce the set of bounded structures, and show how every 3-valued structure can be mapped into a bounded structure. Section 4.5 states the abstract interpretation in terms of a least fixed point of a set of equations.

4.1 The Concrete Stores Represented by a 3-Valued Structure

Definition 4.1. **(Concretization of 3-Valued Structures)** *For a structure $S \in$ 3-STRUCT[$\mathcal{P}$], we denote by $\gamma(S)$ the set of 2-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 Fig. 2 represents two classes of data structures: (i) lists of length two or more, and (ii) lists with one element and one or more garbage cells. The reason that S_2 represents the latter class of data structures is that because $n(u_1, u) = 1/2$, individual u of U^{S_2} may represent elements unreachable from `x` (i.e., uncollected garbage).

It is possible to change the definition of embeddings (abstractions) to exclude garbage cells explicitly (see [Sagiv et al. 1996]). An alternative is to use an additional instrumentation predicate, r, defined by formula (6), to maintain reachability information explicitly. With the latter approach, for every program statement there would be a predicate-update formula to update r. (See Section 6.1.2.)

The structure $S^\natural_{weird}$ shown in Fig. 7 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 and Conditions

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 3-valued structures to 3-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) [Larus and Hilfinger 1988; Chase et al. 1990].

—The (3-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_p^{st} (referred to as p' in Section 2.3).

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

$$[\![st]\!](S) = \langle U^S, \lambda p.\lambda u_1, \dots, u_k.[\![\varphi_p^{st}]\!]_3^S([v_1 \mapsto u_1, \dots, v_k \mapsto u_k])\rangle$$

Table VI 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 Fig. 1(a).

Definition 4.3 does not handle statements of the form `x = malloc()` because the universe of the structure produced by $[\![st]\!](S)$ is the same as the universe of S. Instead, for allocation statements we need to use the modified definition of $[\![st]\!](S)$ given in Definition 4.4, which first allocates a new individual u_{new}, and then invokes predicate-update formulae in a manner similar to Definition 4.3.

Definition 4.4. *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 with** *$st \equiv \mathtt{x} = \mathtt{malloc}()$, denoted by $[\![x = malloc()]\!]$, is defined as follows:*

$$
\begin{array}{l}
[\![\mathtt{x} = \mathtt{malloc}()]\!](S) = \\
\quad \textbf{let } U' = U^S \cup \{u_{new}\}, \textit{ where } u_{new} \textit{ is an individual not in } U^S \\
\quad \textbf{and } \iota' = \lambda p \in (\mathcal{P} \cup \{new\}).\lambda u_1, \dots, u_k. \begin{cases} 1 & p = new \textit{ and } u_1 = u_{new} \\ 0 & p = new \textit{ and } u_1 \neq u_{new} \\ 1/2 & \begin{array}{l} p \neq new \textit{ and there exists } i, \\ 1 \leq i \leq k, \textit{ such that } u_i = u_{new} \end{array} \\ \iota^S(p)(u_1, \dots, u_k) & \text{otherwise} \end{cases} \\
\quad \textbf{in } \langle U', \lambda p \in \mathcal{P}.\lambda u_1, \dots, u_k.[\![\varphi_p^{st}]\!]_3^{\langle U', \iota' \rangle}([v_1 \mapsto u_1, \dots, v_k \mapsto u_k])\rangle
\end{array}
$$

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 = NULL`	$\varphi_z^{st}(v) \stackrel{\text{def}}{=} z(v)$, for each $\mathtt{z} \in PVar$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} n(v_1, v_2) \wedge \neg x(v_1)$ $\varphi_{sm}^{st}(v) \stackrel{\text{def}}{=} sm(v)$
`x->n = t` (assuming `x->n == NULL`)	$\varphi_z^{st}(v) \stackrel{\text{def}}{=} z(v)$, for each $\mathtt{z} \in PVar$ $\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} n(v_1, v_2) \vee (x(v_1) \wedge t(v_2))$ $\varphi_{sm}^{st}(v) \stackrel{\text{def}}{=} sm(v)$
`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 VI. Predicate-update formulae for the core predicates for `List` and `reverse`.

In Definition 4.4, 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 one or more arguments is u_{new}. The predicate-update operation in Definition 4.4 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 4.4 ranges over $\mathcal{P}$.)

3-valued formulae also provide a natural way to define (conservatively) the meaning of program conditions. In particular, we define the meaning of a condition st to be

$$[\![st]\!](S) \stackrel{\text{def}}{=} [\![\varphi^{st}]\!]_3^S([]).$$

(To keep things simple, we assume that conditions do not have side-effects. It is possible to support side-effects in conditions in the same way that is done for statements, namely, by providing appropriate predicate-update formulae.)

—If $[\![\varphi^{st}]\!]_3^S([])$ yields 1, the condition holds in every store represented by S.

—If $[\![\varphi^{st}]\!]_3^S([])$ yields 0, the condition does not hold in any store represented by S.

—If $[\![\varphi^{st}]\!]_3^S([])$ yields 1/2, then we do not know if the condition always holds, never holds, or sometimes holds and sometimes does not hold in the stores represented by S.

st	φ^{st}
`x == y`	$\forall v : x(v) \Leftrightarrow y(v)$
`x != y`	$\exists v : \neg(x(v) \Leftrightarrow y(v))$
`x == NULL`	$\forall v : \neg x(v)$
`x != NULL`	$\exists v : x(v)$
`UninterpretedCondition`	**1/2**

Table VII. 3-valued formulae for conditions involving pointer variables.

3-valued formulae for four types of conditions involving pointer variables are shown in Table VII. Other kinds of conditions involving pointer variables would either have other formulae, or would be handled via the formula for `UninterpretedCondition`.

The Embedding Theorem immediately implies that the 3-valued interpretation is conservative with respect to every store that can possibly occur at run-time.

4.3 Updating the Instrumentation Predicates

Because each instrumentation predicate is defined by means of a formula (cf. Table IV), for the concrete semantics there is no need to specify formulae for updating the instrumentation predicates. However, for the abstract semantics, the Instrumentation Principle implies that it may be more precise for a statement transformer to update the values of the instrumentation predicates. In particular, this is often the case for the instrumentation-predicate value of a summary node, as the following example demonstrates:

EXAMPLE 4.5. Consider the application of statement $st_{5.2}$: `y->n = t` to structure S_7 in Fig. 4. The abstract transformer associated with statement $st_{5.2}$ sets $is'(u)$ to 0 in structure S_8, despite the fact that the value of $\varphi_{is,n}$ at u in S_8, i.e., $[\![\varphi_{is,n}]\!]_3^{S_8}([v \mapsto u])$, is 1/2. This is consistent with the semantics of the statement `y->n = t` because the execution of `y->n = t` can only cause heap cells pointed to by `t` to become shared; because any (concrete) heap cell c represented by u cannot be pointed to by `t`, the (concrete) execution of `y->n = t` cannot make c become shared, and hence $is'(u)$ can be set to 0 in structure S_8.

In order to update the values of the instrumentation predicates based on the stored values of the instrumentation predicates, as part of instantiating the parametric framework, the designer of a shape analysis must provide, for every predicate $p \in \mathcal{I}$ and statement st, a predicate-update formula φ_p^{st} that identifies the new value of p after st. It is always possible to define φ_p^{st} to be the formula $\varphi_p[c \mapsto \varphi_c^{st} \mid c \in \mathcal{C}]$ (i.e., the formula obtained from φ_p by replacing each occurrence of a predicate $c \in \mathcal{C}$ by φ_c^{st}.[3] This substitution captures the value for c after st has been executed. We refer to $\varphi_p[c \mapsto \varphi_c^{st} \mid c \in \mathcal{C}]$ as **the trivial update formula** for predicate p, since it merely reevaluates the p's defining formula in the structure obtained after st has been executed. As demonstrated in Example 4.5, because reevaluation

[3]Here we are making the assumption that the formula for an instrumentation predicate is defined solely in terms of core predicates, and not in terms of other instrumentation predicates. An instrumentation predicate's formula can always be put in this form by repeated substitution until only core predicates occur.

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 = NULL`	$\varphi_{is}^{st}(v) \stackrel{\text{def}}{=} \begin{cases} is(v) \wedge \varphi_{is}[n \mapsto \varphi_n^{st}] & \text{if } \exists v' : x(v') \wedge n(v', v) \\ is(v) & \text{otherwise} \end{cases}$
`x->n = t` (assuming `x->n == NULL`)	$\varphi_{is}^{st}(v) \stackrel{\text{def}}{=} \begin{cases} is(v) \vee \varphi_{is}[n \mapsto \varphi_n^{st}] & \text{if } \exists v_1 : t(v) \wedge n(v_1, v) \\ is(v) & \text{otherwise} \end{cases}$
`x = malloc()`	$\varphi_{is}^{st}(v) \stackrel{\text{def}}{=} is(v) \wedge \neg new(v)$

Table VIII. Predicate-update formulae for the instrumentation predicate is.

may yield many indefinite values, the trivial update formula is often unsatisfactory. It is preferable, therefore, to devise predicate-update formula that minimize reevaluations of φ_p.

Example 4.6. Table VIII gives the predicate-update formulae for the instrumentation predicate is. The assignment to `x->n=NULL` can only change the sharing to false for elements pointed to by `x->n`. Therefore, in Table VIII $\varphi_{is}[n \mapsto \varphi_n^{st}]$ is evaluated only for elements pointed to by `x->n`. Similarly, the assignment `x->n=t` can only change the sharing to true for elements pointed to by `t`, when that element already has at least one incoming edge. Therefore, $\varphi_{is}[n \mapsto \varphi_n^{st}]$ is evaluated only for elements that are pointed to by `t` and already have at least one incoming edge.

We now state the requirements on predicate-update formulae that the user of our framework needs to show in order to make sure that the analysis is conservative.

Definition 4.7. *We say that a predicate-update formula for p* **maintains the correct instrumentation for statement** *st if, for all $S^\natural \in$ 2-CSTRUCT$[\mathcal{P}, F]$ and all Z,*

$$[\![\varphi_p^{st}]\!]_3^{S^\natural}(Z) = [\![\varphi_p]\!]_3^{[\![st]\!](S^\natural)}(Z). \tag{42}$$

In the above definition, $[\![st]\!](S^\natural)$ denotes a version of the operation defined in Definitions 4.3 and 4.4 in which $\mathcal{P}$ is restricted to $\mathcal{C}$. (Here we are making the assumption that the predicate-update formula for an instrumentation predicate is defined solely in terms of core predicates, and not in terms of instrumentation predicates.)

In the sequel, we assume that for all the instrumentation predicates and all the statements, the predicate-update formulae maintain correct instrumentation. Note that the trivial update formuae do maintain correct instrumentation; however, they may yield very imprecise answers when applied to 3-valued structures.

4.4 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 $\iota^S(p)(u_1) \neq \iota^S(p)(u_2)$.*

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

The consequence of Definition 4.8 is that 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., $|U^S| \leq 3^{|\mathcal{A}|}$.

EXAMPLE 4.9. Consider the class of bounded structures associated with the `List` data-type declaration from Fig. 1(a). Here the predicate symbols are $\mathcal{C} = \{sm, n\} \cup \{x \mid \mathtt{x} \in PVar\}$ and $\mathcal{I} = \{is\}$. Notice that the sm predicate plays a different role than other core predicates since it captures the information lost in the abstraction, and has a trivial fixed meaning of 0 in all concrete structures. (We choose to include sm in the concrete structures to avoid the need to work with different vocabularies at the concrete and abstract levels.)

For the `reverse` program from Fig. 1(b), the program variables are `x`, `y`, and `t`, yielding unary core predicates sm, x, y, and t; the only other unary predicate is is. 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, Fig. 4 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 3-valued structure, not just 2-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 Fig. 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 Fig. 4, 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 is isomorphic to S', 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 predicates $\iota^S(p)$ and $\iota^{S'}(p)$ are equal.

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 [Sagiv et al. 1998].[4] The only abstraction predicates used in [Sagiv et al. 1998] are the "pointed-to-by-variable-x" predicates. Because sharing predicate is is used only as an instrumentation predicate in [Sagiv et al. 1998], but not as an abstraction predicate, the algorithm from [Sagiv et al. 1998] does not distinguish between shared and unshared individuals, and thus loses precision 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 precision 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, which is important for maintaining precision when analyzing, for instance, a program that swaps the first two elements of a list.

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.

In [Wang 1994; Sagiv et al. 1998], unbounded-size stores are embedded into bounded-size abstractions by collapsing 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 [Jones and Muchnick 1981]. Definition 4.10 generalizes these ideas 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 3-valued logical structures, we are able to apply t_embed_c to abstract (3-valued) structures as well as to concrete (2-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.

[4]The shape-analysis algorithm presented in [Sagiv et al. 1998] 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.

—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 3-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.

More generally, the idea developed in this paper of using 3-valued logic for program analysis has broader applicability than just the shape-analysis problem. Applications of the machinery developed in this paper to other program-analysis problems are discussed in [Nielson et al. 1999; Lev-Ami et al. 2000].

4.5 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.)

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

The shape-analysis algorithm itself is an iterative procedure. For each vertex v of control-flow graph G, it computes a set $StructSet[v]$ of structures that hold on entry to v as a least fixed point of the following system of equations (over the variables $StructSet[v]$):

$$StructSet[v] = \begin{cases} \{\langle\emptyset,\emptyset\rangle\} & \text{if } v = start \\ \left.\begin{array}{l} StructSet[v] \cup \bigcup\limits_{\substack{w \to v \in E(G),\\ w \in As(G)}} \{t\_embed_c[\![st(w)]\!](S) \mid S \in StructSet[w]\} \\ \quad\cup \bigcup\limits_{\substack{w \to v \in E(G),\\ w \in Id(G)}} \{S \mid S \in StructSet[w]\} \\ \quad\cup \bigcup\limits_{w \to v \in Tb(G)} \{S \mid S \in StructSet[w] \text{ and } [\![st(w)]\!](S) \sqsupseteq 1\} \\ \quad\cup \bigcup\limits_{w \to v \in Fb(G)} \{S \mid S \in StructSet[w] \text{ and } [\![st(w)]\!](S) \sqsupseteq 0\} \end{array}\right\} & \text{if } v \neq start \end{cases} \tag{43}$$

In equation (43), $As(G)$ denotes the set of assignment statements that manipulate pointers; $Id(G)$ denotes the set of assignment statements that do not manipulate pointers (these statements are uninterpreted); $Tb(G) \subseteq E(G)$ and $Fb(G) \subseteq E(G)$ are the subsets of the G's edges that represent true and false branches from conditions, respectively.

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.

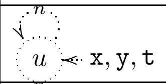
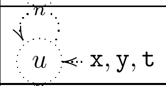
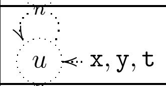
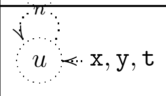
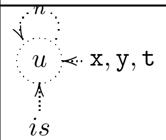
statement	formula	structure	
st_2: `t = y;`	$t'(v) = y(v)$		S'_{14}
st_3: `y = x;`	$y'(v) = x(v)$		S'_{15}
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$		S'_{16}
$st_{5,1}$: `y->n = NULL;`	$n'(v_1, v_2) = n(v_1, v_2) \wedge \neg y(v_1)$ $is'(v) = \begin{cases} is(v) \wedge \varphi_{is,n'} & \text{if } \exists v' : y(v') \wedge n(v', v) \\ is(v) & \text{otherwise} \end{cases}$		S'_{17}
$st_{5.2}$: `y->n = t;`	$n'(v_1, v_2) = n(v_1, v_2) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = \begin{cases} is(v) \vee \varphi_{is,n'} & \text{if } \exists v_1 : t(v) \wedge n(v_1, v) \\ is(v) & \text{otherwise} \end{cases}$		S'_{18}

Table IX. The bounded structures that actually arise for the last four blocks of Fig. 4 when t_embed_c is applied at each step.

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 five blocks of Fig. 4 are deliberately inconsistent with equation (43). The bounded structures that actually arise when t_embed_c is applied at each step are shown in Table IX. In structures S'_{14}, S'_{15}, S'_{16} and S'_{17}, the canonical name of the one individual u is $u_{\emptyset,\{is\}}$. In structure S'_{18}, the canonical name of individual u is $u_{\emptyset,\emptyset}$.[5]

Other variations on equation (43) 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.

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 Fig. 9 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 Fig. 8 and explained below:

(1) The operation *focus*, defined in Section 5.1, refines 3-valued structures so that

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

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

Fig. 8. One- vs. three-stage abstract semantics of statement st_3. The notation $\llbracket st \rrbracket$ denotes the operation defined in Section 4.2. 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.)

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 4.4).

(3) The operation *coerce*, defined in Section 5.2, converts a 3-valued structure into a more precise 3-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 (a concept originally introduced in [Cousot and Cousot 1979]). That is, they convert a set of 3-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. In the context of a parametric framework for abstract interpretation, semantic reductions are valuable because they allow the transformers of the abstract semantics to be defined in a modular fashion.

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 generates a set of structures on which a given set of formulae F have definite values for all individuals. Unfortunately, in general *focus* may yield an infinite set of structures. Therefore, in Section 5.1.1, we declaratively specify the properties of the *focus* operation, and in Section 5.1.2, we give an algorithm that implements *focus* for a certain specific class of formulae that are needed for shape analysis. The latter algorithm always yields a finite set of structures.

5.1.1 *The Focus Operation.* 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, $\llbracket st \rrbracket$, etc.),

$$\widehat{op}(XS) \stackrel{\text{def}}{=} \bigcup_{S \in XS} op(S). \tag{44}$$

Definition 5.1. *Given a set of formulae F, a function op: 3-STRUCT$[\mathcal{P}] \to 2^{\textit{3-STRUCT}[\mathcal{P}]}$ is* **a focus operation for** *F if for every $S \in$ 3-STRUCT$[\mathcal{P}]$, $op(S)$ satisfies the following requirements:*

—*$op(S)$ and S represent the same concrete structures, i.e., $\gamma(S) = \widehat{\gamma}(op(S))$*

—*Every formula $\varphi \in F$ has a definite value in each of the structures in $op(S)$, i.e., for every formula $\varphi \in F$, $S' \in op(S)$, and assignment Z, we have $[\![\varphi]\!]_3^{S'}(Z) \neq 1/2$.*

Henceforth, we will use the notation $focus_F$, or simply *focus* when F is clear from the context, when referring to a focus operation for F in the generic sense. We will consider a specific algorithm for focusing shortly.

The first obstacle to developing a general algorithm for focusing is that the number of resulting structures may be infinite. In many cases (including the ones used below for shape analysis), this can be overcome by only generating maximal structures. However, in some cases the set of maximal structures is infinite, as well. This phenomenon is illustrated by the following example:

EXAMPLE 5.2. Consider the following formula

$$\varphi_{last}(v) \stackrel{\text{def}}{=} \forall v_1 : \neg n(v, v_1),$$

which is true for the last heap cell of an acyclic singly linked list. Focusing on φ_{last} with the structure S_2 shown in Fig. 2 will lead to an infinite set of maximal structures (of length 1, 2, 3, etc.)

To sidestep this obstacle, the focus formulae φ used in shape-analysis are determined by the L-values and R-values of each kind of statement in the programming language. These are formally defined in Section 5.1.2 and illustrated in the following example.

EXAMPLE 5.3. For the statement st_4: `x = x->n` in procedure `reverse`, we focus on the formula

$$\varphi_0(v) \stackrel{\text{def}}{=} \exists v_1 : x(v_1) \wedge n(v_1, v), \tag{45}$$

which corresponds to the R-value of st_4 (the heap cell pointed to by `x->n`). The upper part of Fig. 9 illustrates the application of $focus_{\{\varphi_0\}}(S_5)$, where S_5 is the structure shown in Fig. 4 that we have in `reverse` just before the first application of statement st_4: `x = x->n`. This results in three structures: $S_{5,f,0}$, $S_{5,f,1}$, and $S_{5,f,2}$.

—In $S_{5,f,0}$, $[\![\varphi_0]\!]_3^{S_{5,f,0}}([v \mapsto u])$ equals 0. 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.

—In $S_{5,f,1}$, $[\![\varphi_0]\!]_3^{S_{5,f,1}}([v \mapsto u])$ equals 1. This covers the case where the list that `x` and `y` point to has exactly two elements: For all of the concrete cells that summary node u represents, φ_0 must evaluate to 1, and so u must represent just a single list node.

—In $S_{5,f,2}$, $[\![\varphi_0]\!]_3^{S_{5,f,2}}([v \mapsto u.0])$ equals 0 and $[\![\varphi_0]\!]_3^{S_{5,f,2}}([v \mapsto u.1])$ equals 1. This covers the case where the list that `x` and `y` point to is a list of three or more elements: For all of the concrete cells that $u.0$ represents, φ_0 must evaluate to 0, and for all of the cells that $u.1$ represents, φ_0 must evaluate to 1. This case captures the essence of node materialization as described in [Sagiv et al. 1998]: individual u is bifurcated into two individuals.

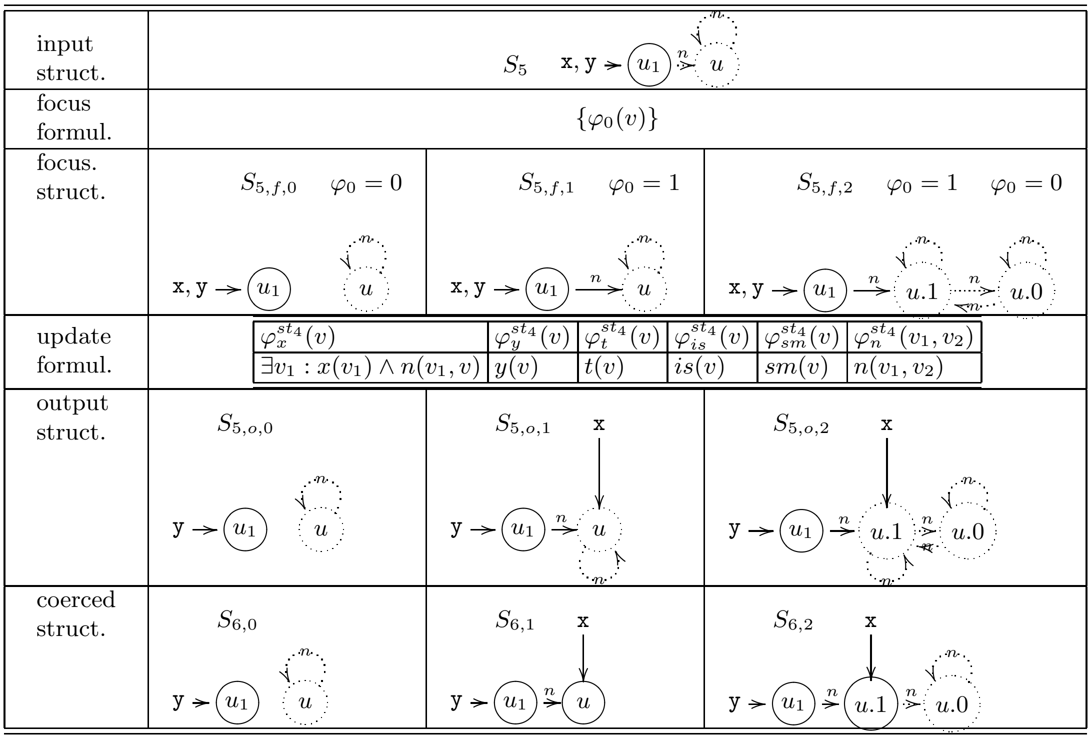


Fig. 9. The first application of the improved transformer for statement st_4: `x = x->n` in `reverse`.

Notice how $focus_{\{\varphi_0\}}(S_5)$ can be effectively constructed from S_5 by considering the reasons why $[\![\varphi_0]\!]_3^{S_5}(Z)$ evaluates to 1/2 for an assignment Z. In some cases, $[\![\varphi_0]\!]_3^{S_5}(Z)$ already has a definite value; for instance $[\![\varphi_0]\!]_3^{S_5}([v \mapsto u_1])$ equals 0, and therefore φ_0 is already in focus at u_1. In contrast, $[\![\varphi_0]\!]_3^{S_5}([v \mapsto u])$ equals 1/2. There are three (maximal) structures S that we can construct from S_5 in which $[\![\varphi_0]\!]_3^{S}([v \mapsto u])$ has a definite value:

—$S_{5,f,0}$, in which $n(u_1, u)$ was forced to 0, and thus $[\![\varphi_0]\!]_3^{S_{5,f,0}}([v \mapsto u])$ equals 0.

—$S_{5,f,1}$, in which $n(u_1, u)$ was forced to 1, and thus $[\![\varphi_0]\!]_3^{S_{5,f,1}}([v \mapsto u])$ 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 $[\![\varphi_0]\!]_3^{S_{5,f,2}}([v \mapsto u.0])$ equals 0, whereas $n(u_1, u.1)$ was set to 1, and thus $[\![\varphi_0]\!]_3^{S_{5,f,2}}([v \mapsto u.1])$ equals 1.

Of course, there are other structures that can be embedded into S_5 that would assign a definite value to φ_0, but these are not maximal because each of them can be embedded into one of $S_{5,f,0}$, $S_{5,f,1}$, or $S_{5,f,2}$.

5.1.2 *Selecting the Set of Focus Formulae For Shape Analysis.* The greater the number of formulae on which we focus, the greater the number of distinctions that the shape-analysis algorithm can make, leading to improved precision. However,

st	Focus Formulae
`x = NULL`	$\emptyset$
`x = t`	$\{t(v)\}$
`x = t->n`	$\{\exists v_1 : t(v_1) \wedge n(v_1, v)\}$
`x->n = t`	$\{x(v), t(v)\}$
`x = malloc()`	$\emptyset$
`x == NULL`	$\emptyset$
`x != NULL`	$\emptyset$
`x == t`	$\{x(v), t(v)\}$
`x != t`	$\{x(v), t(v)\}$
`UninterpretedCondition`	$\emptyset$

Table X. The target formulae for *focus*, for statements and conditions of a program that uses type `List`.

using a larger number of focus formulae can increase the number of structures that arise, thereby increasing the cost of analysis. Our preliminary experience indicates that in shape analysis there is a simple way to define the formulae on which to focus that guarantees that the number of structures generated grows only by a constant factor. The main idea is that in a statement of the form $lhs = rhs$, we only focus on formulae that define the heap cells for the L-value of lhs and the R-value of rhs. This definition extends naturally to program conditions and to statements that manipulate multiple L- and R-values.

For our simplified language and type `List`, the target formulae on which to focus can be defined as shown in Table X. Let us examine a few of the cases from Table X:

—For the statement `x = NULL`, the set of target formulae is the empty set because neither the L-value nor the R-value is a heap cell.

—For the statement `x = t->n`, the set of target formulae is the singleton set $\{\exists v_1 : t(v_1) \wedge n(v_1, v)\}$ because the L-value cannot be a heap cell, and the R-value is a cell pointed to by `t->n`.

—For the statement `x->n = t`, the set of target formulae is the set $\{x(v), t(v)\}$ because the L-value is a heap cell pointed to by `x` and the R-value is a heap cell pointed to by `t`.

—For the condition `x == t`, the set of target formulae is the set $\{x(v), t(v)\}$: there are no L-values, and the R-values of the statement are the heap cells pointed to by `x` and `t`.

It is not hard to extend Table X for statements that manipulate more complicated data structures involving chains of selectors. For example, the set of target formulae for the statement `x->a->b = y->c->d->e` is

$$\{\exists v_1 : x(v_1) \wedge a(v_1, v), \exists v_1, v_2, v_3 : y(v_1) \wedge c(v_1, v_2) \wedge d(v_2, v_3) \wedge e(v_3, v)\},$$

because the L-value is a heap cell pointed to by `x->a`, and the R-value is a heap cell pointed to by `y->c->d->e`.

Fig. 10 contains an algorithm that implements *focus* for the type of formulae that arise in Table X. We observe that for every set of formulae $F_1 \cup F_2$, it is possible to focus on $F_1 \cup F_2$ by first focusing on F_1, and then on F_2. Thus, it is sufficient to provide an algorithm that focuses on an individual formula φ. Also, there are two types of formulae used in Table X:

—The formula $\varphi \equiv x(v)$, for $\mathtt{x} \in PVar$. In this case, FocusVar($S, \mathtt{x}$) is applied.

—The formula $\varphi \equiv \exists v_1 : x(v_1) \wedge n(v_1, v)$, for $\mathtt{x} \in PVar$. In this case, FocusVarDeref($S, \mathtt{x}, n$) is applied.

FocusVar repeatedly eliminates more and more indefinite values for $x(v)$ by creating more and more structures. For every individual u for which $\iota^S(x)(u)$ is an indefinite value, two or three structures are created. The function Expand creates a structure in which individual u is bifurcated into two individuals; this captures the essence of shape-node materialization (cf. [Sagiv et al. 1998]).

FocusVarDeref first brings $x(v)$ into focus (by invoking FocusVar), and then proceeds to eliminate indefinite $\iota^S(n)$ values.

Example 5.4. Consider the application of FocusVarDeref($S_5, \mathtt{x}, n$) for S_5 defined in Fig. 9. In this case, FocusVar($S_5, \mathtt{x}$) $= \{S_5\}$. When S_5 is selected from the worklist, structures $S_{5,f,0}$, $S_{5,f,1}$, and $S_{5,f,2}$ are created. In the next three iterations, these structures are moved to `AnswerSet`.

The following two lemmas guarantee that the algorithm for *focus* shown in Fig. 10 is correct.

Lemma 5.5. *For $\varphi \equiv x(v)$, and for every structure $S \in$ 3-STRUCT[$\mathcal{P}$], $FocusVar(S, \mathtt{x})$ is a focus operation for $\{\varphi\}$.*

Lemma 5.6. *For $\varphi \equiv \exists v_1 : x(v_1) \wedge n(v_1, v)$, and for every structure $S \in$ 3-STRUCT[$\mathcal{P}$], $FocusVarDeref(S, \mathtt{x}, \mathtt{n})$ is a focus operation for $\{\varphi\}$.*

It is not hard to see that both FocusVar and FocusVarDeref always return a finite set of structures.

In what follows, $Focus_\varphi$ denotes the operation that invokes FocusVar or FocusVarDeref, as appropriate.

5.2 Coercing into More Precise Structures

After *focus*, we apply the simple transformer $[\![st]\!]$ that was defined in Definitions 4.3 and 4.4. In the example discussed in Section 5.1, we apply $[\![st_4]\!]$ to the structures $S_{5,f,0}$, $S_{5,f,1}$, and $S_{5,f,2}$. Structure $S_{5,o,0}$ 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 5.7. Consider structure $S_{5,o,2}$ from Fig. 9. 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 $u.0$.

function FocusVar(S_0 : 3-STRUCT[$\mathcal{P}$], $\mathtt{x}$: $PVar$) **returns** $2^{\text{3-STRUCT}[\mathcal{P}]}$
begin
  WorkSet := $\{S_0\}$
  AnswerSet := $\emptyset$
  while WorkSet $\neq \emptyset$ **do**
    Select and remove a structure S from WorkSet
    if there exists $u \in U^S$ s.t. $\iota^S(x)(u) = 1/2$ **then**
     Insert $\langle U^S, \iota^S[x(u) \mapsto 0]\rangle$ into WorkSet
     Insert $\langle U^S, \iota^S[x(u) \mapsto 1]\rangle$ into WorkSet
     if $\iota^S(sm)(u) = 1/2$ **then**
      let $u.0$ and $u.1$ be individuals not in U^S
      and $S' = \text{Expand}(S, u, u.0, u.1)$
      Insert $\langle U^{S'}, \iota^{S'}[x(u.0) \mapsto 0, x(u.1) \mapsto 1]\rangle$ into WorkSet
     fi
    else
     Insert S into AnswerSet
    fi
  od
  return AnswerSet
end

function FocusVarDeref(S_0 : 3-STRUCT[$\mathcal{P}$], $\mathtt{x}$: $PVar$, $\mathtt{n}$:Selector) **returns** $2^{\text{3-STRUCT}[\mathcal{P}]}$
begin
  WorkSet := FocusVar($S_0, \mathtt{x}$)
  AnswerSet := $\emptyset$
  while WorkSet $\neq \emptyset$ **do**
    Select and remove a structure S from WorkSet
    if there exists $u_1, u \in U^S$ s.t. $\iota^S(x)(u_1) = 1$ and $\iota^S(n)(u_1, u) = 1/2$ **then**
     Insert $\langle U^S, \iota^S[n(u_1, u) \mapsto 0]\rangle$ into WorkSet
     Insert $\langle U^S, \iota^S[n(u_1, u) \mapsto 1]\rangle$ into WorkSet
     if $\iota^S(sm)(u) = 1/2$ **then**
      let $u.0$ and $u.1$ be individuals not in U^S
      and $S' = \text{Expand}(S, u, u.0, u.1)$
      Insert $\langle U^{S'}, \iota^{S'}[n(u_1, u.0) \mapsto 0, n(u_1, u.1) \mapsto 1]\rangle$ into WorkSet
     fi
    else
     Insert S into AnswerSet
    fi
  od
  return AnswerSet
end
function Expand(S : 3-STRUCT[$\mathcal{P}$], $u, u.0, u.1$: elements)
   returns 3-STRUCT[$\mathcal{P}$]
let $m = \lambda u'. \begin{cases} u & \text{if } u' = u.0 \vee u' = u.1 \\ u' & \text{otherwise} \end{cases}$ **in**
  return $\left\langle \begin{array}{l} (U^S - \{u\}) \cup \{u.0, u.1\} \\ \lambda p.\lambda u_1, \ldots, u_k.\iota^S(p)(m(u_1), \ldots, m(u_k)) \end{array} \right\rangle$

Fig. 10. An algorithm for *focus* for the two types of formulae that arise in Table X.

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 5.7. 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 2-valued structures are just interpreted as 3-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 3-valued structures:

EXAMPLE 5.8. Consider a 2-valued structure $S^\natural$ that can be embedded in a 3-valued structure S, and suppose that the formula φ_{is} for "inferring" whether an individual u is shared evaluates to 1 in S (i.e., $[\![\varphi_{is}(v)]\!]_3^S([v \mapsto u]) = 1$). By the Property-Extraction Principle (Observation 2.1), $is(u^\natural)$ must be 1 for any individual $u^\natural \in U^{S^\natural}$ that the embedding function maps to u.

Now consider a structure S' that is equal to S except that $is(u)$ is $1/2$. $S^\natural$ can also be embedded in S'. However, the embedding of $S^\natural$ in S is a "better" embedding; it is a "tighter embedding" in the sense of Definition 3.9. This has operational significance: It is needlessly imprecise to work with structure S' in which $is(u)$ has the value $1/2$; instead, we should discard S' and work with S. In general, the "stored property" is should be at least as precise as its inferred value; consequently, if it happens that φ_{is} evaluates to a definite value (1 or 0) in a 3-valued structure, we can 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 3-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 5.8 can be summarized as the following principle:

OBSERVATION 5.9. [**The Sharpening Principle**]. *In any structure S, the value 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 3-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 5.10. *Let Σ be a finite set of compatibility constraints of the form $\varphi_1 \rhd \varphi_2$, where φ_1 is an arbitrary 3-valued formula, and φ_2 is either an atomic formula or the negation of an atomic formula. We say that a structure S* **satisfies** *Σ (denoted by $S \models \Sigma$) if for every constraint $\varphi_1 \rhd \varphi_2$ in Σ, and for every assignment Z such that $[\![\varphi_1]\!]_3^S(Z) = 1$, we have $[\![\varphi_2]\!]_3^S(Z) = 1$.*

For a 2-valued structure, $\rhd$ has the same meaning as implication. (That is, if S is a 2-valued structure, $S, Z \models \varphi_1 \rhd \varphi_2$ iff $S, Z \models \varphi_1 \Rightarrow \varphi_2$.) However, for a 3-valued structure, $\rhd$ is stronger than implication: if φ_1 evaluates to 1 and φ_2 evaluates to 1/2, the constraint $\varphi_1 \rhd \varphi_2$ is not satisfied. More precisely, suppose that $[\![\varphi_1]\!]_3^S(Z) = 1$ and $[\![\varphi_2]\!]_3^S(Z) = 1/2$; the implication $\varphi_1 \Rightarrow \varphi_2$ is satisfied (i.e., $S, Z \models \varphi_1 \Rightarrow \varphi_2$), but the constraint $\varphi_1 \rhd \varphi_2$ is not satisfied (i.e., $S, Z \not\models \varphi_1 \rhd \varphi_2$).

The constraint that captures the reasoning used in Example 5.8 is $\varphi_{is}(v) \rhd is(v)$. That is, when φ_{is} evaluates to 1 at u, then is must evaluate to 1 at u.

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

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 $\rhd$). The reason is that formulae are monotonic in the information order (see Lemma 3.7), whereas $\rhd$ is non-monotonic in its right-hand-side argument: The constraint $\mathbf{1} \rhd \mathbf{1}$ 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 $\mathbf{1} \rhd \mathbf{1/2}$ is one that is not satisfied in any structure.

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

DEFINITION 5.11. *For formula φ and atomic formula a with free variables v_1, v_2, ..., v_k (such that $a \not\equiv sm$) we generate a constraint r as follows.*

$$r(\forall v_1, \ldots v_k : (\varphi \Rightarrow a)) \stackrel{\text{def}}{=} \varphi \rhd a \tag{46}$$

$$r(\forall v_1, \ldots v_k : (\varphi \Rightarrow \neg a)) \stackrel{\text{def}}{=} \varphi \rhd \neg a \tag{47}$$

$$r(\forall v_1, \ldots v_k : \varphi) \stackrel{\text{def}}{=} \neg\varphi \rhd \mathbf{0} \tag{48}$$

For a set of formulae F, we define $\widehat{r}(F)$ to be the set of constraints obtained by applying r to each of the formulae in F.

The intuition behind (46) and (47) is that for an atomic predicate, a tight embedding yields 1/2 only in cases in which a evaluates to 1 on one tuple of values for $v_1, \ldots v_k$, but evaluates to 0 on a different tuple of values. In this case, the left-hand side will evaluate to 1/2 as well (see Lemma 5.13 below). Rule (48) was added to enable an arbitrary formula to be converted to a constraint.

EXAMPLE 5.12. The constraints generated for the formulae that appear above the line in Table V are listed above the line in Table XI.

The following Lemma guarantees that tight embedding preserves satisfaction of $\widehat{r}(F)$.

LEMMA 5.13. *For every pair of structures $S^\natural \in$ 2-CSTRUCT$[\mathcal{P}, F]$ and $S \in$ 3-STRUCT$[\mathcal{P}]$ such that S is a tight embedding of $S^\natural$, $S \models \widehat{r}(F)$.*

$sm(v)$	$\rhd$	$\mathbf{0}$	(49)
for each $\mathtt{x} \in PVar, x(v_1) \wedge x(v_2)$	$\rhd$	$v_1 = v_2$	(50)
$(\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2))$	$\rhd$	$v_1 = v_2$	(51)
$(\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2)$	$\rhd$	$is(v)$	(52)
$\neg(\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2)$	$\rhd$	$\neg is(v)$	(53)
for each $\mathtt{x} \in PVar, (\exists v_1 : x(v_1) \wedge v_1 \neq v_2)$	$\rhd$	$\neg x(v_2)$	(54)
for each $\mathtt{x} \in PVar, (\exists v_2 : x(v_2) \wedge v_1 \neq v_2)$	$\rhd$	$\neg x(v_1)$	(55)
$(\exists v_1 : n(v_3, v_1) \wedge v_1 \neq v_2)$	$\rhd$	$\neg n(v_3, v_2)$	(56)
$(\exists v_2 : n(v_3, v_2) \wedge v_1 \neq v_2)$	$\rhd$	$\neg n(v_3, v_1)$	(57)
$(\exists v_1 : \neg is(v) \wedge n(v_1, v) \wedge v_1 \neq v_2)$	$\rhd$	$\neg n(v_2, v)$	(58)
$(\exists v_2 : \neg is(v) \wedge n(v_2, v) \wedge v_1 \neq v_2)$	$\rhd$	$\neg n(v_1, v)$	(59)
$(\exists v : \neg is(v) \wedge n(v_1, v) \wedge n(v_2, v))$	$\rhd$	$v_1 = v_2$	(60)

Table XI. The compatibility constraints $\widehat{r}(\widehat{closure}(F))$ generated using Definitions 5.11 and 5.15 from the formulae F given above the line in Table V (i.e., formulae (29)–(32)). Formulae (33)–(39), given below the line in Table V, are $\widehat{closure}(F)$. Constraints (54)–(60) come from applying r to formulae (33)–(39) from Table V.

Proof: See Appendix B.

EXAMPLE 5.14. It is worthwhile to notice that tight embedding need not preserve implications when the right-hand side is an arbitrary formula. In particular, it does not hold for disjunctions. Consider the implication formula

$\forall v : 1 \Rightarrow p_1(v) \vee p_2(v)$

and the structure $S^\natural = \langle \{u_1, u_2\}, \iota^\natural \rangle$ with two individuals, u_1 and u_2, such that

$$\iota^\natural = \begin{bmatrix} sm \mapsto [u_1 \mapsto 0, u_2 \mapsto 0], \\ p_1 \mapsto [u_1 \mapsto 1, u_2 \mapsto 0], \\ p_2 \mapsto [u_1 \mapsto 0, u_2 \mapsto 1] \end{bmatrix}$$

Let S be the tight embedding of $S^\natural$ obtained by mapping both u_1 and u_2 into the same individual $u_{1,2}$; that is, $S = \langle \{u_{1,2}\}, \iota \rangle)$ and

$$\iota = \begin{bmatrix} sm \mapsto [u_{1,2} \mapsto 1/2], \\ p_1 \mapsto [u_{1,2} \mapsto 1/2], \\ p_2 \mapsto [u_{1,2} \mapsto 1/2] \end{bmatrix}$$

We see that $S^\natural \models \forall v : 1 \Rightarrow p_1(v) \vee p_2(v)$ but $S \not\models 1 \rhd p_1(v) \vee p_2(v)$ since $[\![p_1(v) \vee p_2(v)]\!]_3^S([v \mapsto u_{1,2}]) = 1/2$, whereas $[\![\mathbf{1}]\!]_3^S([v \mapsto u_{1,2}]) = 1$.

The constraint-generation rules defined in Definition 5.11 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 V 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, $r(\forall v_1, \ldots v_k : (\varphi \Rightarrow a))$ yields the (useful) constraint $\varphi \rhd a$, but $r(\forall v_1, \ldots v_k : (\neg\varphi \vee a))$ yields the (not useful) constraint $\neg(\neg\varphi \vee a) \rhd \mathbf{0}$.

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 5.15 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 5.15. *For a formula φ, we define $\varphi^1 \equiv \varphi$ and $\varphi^0 \equiv \neg\varphi$. We say that a formula φ of the form*

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

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

$$closure(\varphi) \stackrel{\text{def}}{=} \left\{ \forall \ldots, \exists v_1, v_2, \ldots, v_n : \bigwedge_{i=1, i \neq j}^{m} \varphi_i^{1-B_i} \Rightarrow \varphi_j^{B_j} \;\middle|\; \begin{array}{l} 1 \leq j \leq m, \\ v_k \in freeVars(\varphi), \\ v_k \notin freeVars(\varphi_j) \end{array} \right\} \quad (61)$$

For a formula φ that is not an extended Horn clause, $closure(\varphi) = \{\varphi\}$. 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 5.16. Table XI shows the compatibility constraints obtained from $\widehat{r}(\widehat{closure}(F))$, where F is the set of formulae given above the line in Table V (i.e., formulae (29)–(32)). For the extended Horn clauses given as formulae (29)–(32), the new formulae generated by *closure* are listed below the line in Table V (i.e., formulae (33)–(39)). The constraints generated from these formulae are listed below the line in Table XI (cf. (54)–(60)).

In particular, computability formula (31) is

$$\forall v : (\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \Rightarrow is(v)$$

Expressing the implication as a disjunction, we have

$$\forall v : \neg(\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \vee is(v)$$

which can be rewritten by using De Morgan laws as the following extended Horn clause:

$$\forall v, v_1, v_2 : v_1 = v_2 \vee \neg n(v_1, v) \vee \neg n(v_2, v) \vee is(v).$$

From this, we obtain compatibility formulae (37), (38), and (39), which by Definition 5.11, yield compatibility constraints (58), (59) and (60).

As we will see in Section 5.2.3, 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 (58) (or the equivalent constraint (59)) allows us to do a more accurate job of materialization: When $is(u)$ is 0 and one incoming n edge to u is 1, to satisfy constraint (58) a second incoming n edge to u cannot have the value 1/2—it must have the value 0, i.e., the latter edge cannot exist (cf. Examples 5.7 and 5.22). 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 Fig. 9), 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 5.17. **(Compatible 3-Valued Structures)**. *Given a set of hygiene conditions F, the* **set of compatible 3-valued structures** *$3\text{-}CSTRUCT[\mathcal{P}, \widehat{r}(F)] \subseteq 3\text{-}STRUCT[\mathcal{P}]$ is defined by $S \in 3\text{-}CSTRUCT[\mathcal{P}, \widehat{r}(F)]$ iff $S \models \widehat{r}(F)$.*

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

Lemma 5.18. *For every structure $S \in 3\text{-}STRUCT[\mathcal{P}]$ and concrete structure $S^\natural \in \gamma(S)$, there exists a structure $S' \in 3\text{-}CSTRUCT[\mathcal{P}, \widehat{r}(F)]$ 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 $f\colon U^{S^\natural} \to U^S$ such that $S^\natural \sqsubseteq^f S$. Define $S' = f(S^\natural)$ (i.e., S' is the tight embedding of $S^\natural$ under f). By Lemma 5.13, S' satisfies the necessary requirements.

In Section 5.2.3, we give an algorithm that constructs from S and $\widehat{r}(F)$ a maximal S' meeting the conditions of Lemma 5.18 (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 5.19. Consider structure $S_{5,o,2}$ from Fig. 9. This structure violates constraint (58) under the assignment $[v \mapsto u.1, v_1 \mapsto u_1, v_2 \mapsto u.0]$. That is, 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 (58) 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 5.20. *The operation*

$coerce\colon 3\text{-}STRUCT[\mathcal{P}] \to 3\text{-}CSTRUCT[\mathcal{P}, \widehat{r}(F)] \cup \{\bot\}$

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

It is a fact that the maximal such structure S' is unique (if it exists), which follows from the observation that consistent structures (i.e., those with the same universe of individuals) are closed under the following join operation:

DEFINITION 5.21. *For every pair of structures $S_1, S_2 \in$ 3-CSTRUCT$[\mathcal{P}, \widehat{r}(F)]$ such that $U^{S_1} = U^{S_2} = U$, the* **join** *of S_1 and S_2, denoted by $S_1 \sqcup S_2$, is defined as follows:*

$$S_1 \sqcup S_2 \stackrel{\text{def}}{=} \langle U, \lambda p.\lambda u_1, u_2, \ldots, u_m.\iota^{S_1}(p)(u_1, u_2, \ldots, u_m) \sqcup \iota^{S_2}(p)(u_1, u_2, \ldots, u_m)\rangle.$$

In Lemma B.1 (Appendix B) we show that consistent structures are closed under join. Because of uniqueness of the resultant structure, *coerce* is not defined in terms of a set former—in contrast to *focus*, which can return a non-singleton set. The significance of this is that only *focus* can increase the number of structures that arise during shape analysis, whereas *coerce* cannot.

EXAMPLE 5.22. 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 Fig. 9. 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 Table XI.

—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 (58) 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 (50) 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 (58) 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 (58) 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 (50) 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 Fig. 4). For instance, 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, called Coerce, that implements the operation *coerce*. This algorithm actually finds a maximal solution to a system of constraints of the form defined in Definition 5.10. It is convenient to partition these constraints into the following types:

$$\varphi(v_1, v_2, \ldots, v_k) \;\triangleright\; \boldsymbol{b} \tag{62}$$

$$\varphi(v_1, v_2, \ldots, v_k) \;\triangleright\; (v_1 = v_2)^b \tag{63}$$

$$\varphi(v_1, v_2, \ldots, v_k) \;\triangleright\; p^b(v_1, v_2, \ldots, v_k) \tag{64}$$

where $p \neq sm$, $\boldsymbol{b} \in \{\mathbf{0}, \mathbf{1}\}$, and the superscript notation used is the same as in Definition 5.15: $\varphi^1 \equiv \varphi$ and $\varphi^0 \equiv \neg\varphi$. We say that constraints in the forms (62), (63), and (64) are *Type I*, *Type II*, and *Type III* constraints, respectively.

The Coerce algorithm is shown in Fig. 11. The input is a 3-valued structure $S \in$ 3-STRUCT$[\mathcal{P}]$ and a set of constraints $\widehat{r}(F)$. It initializes S' to the input structure S and then repeatedly refines S' by lowering predicate values in $\iota^{S'}$ from 1/2 to a definite value, until either: (i) a constraint is irreparably violated, i.e., the left-hand side and the right-hand side have different definite values, in which case the algorithm fails and returns $\perp$, or (ii) no constraint is violated, in which case the algorithm succeeds and 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 when the right-hand side is a negated equality cannot be fixed: When $v_1 \neq v_2$ does not evaluate to 1, we have $Z(v_1) = Z(v_2)$; therefore, it is impossible to lower predicate values to force the formula $v_1 \neq v_2$ to evaluate to 1 for assignment Z.

—A violation of a Type II constraint having the right-hand side is an 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 right-hand-side value is 1/2.

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

THEOREM 5.23. *For every $S \in$ 3-STRUCT$[\mathcal{P}]$, coerce$(S) =$* Coerce(S).
Proof: See Appendix B.

6. INSTRUMENTATION PREDICATES

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.

In this section, we demonstrate this by defining some interesting instrumentation predicates and showing their significance for shape analysis. In Section 6.1,

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

Fig. 11. An iterative algorithm for solving 3-valued constraints.

we discuss instrumentation predicates that track reachability properties. In Section 6.2, we define instrumentation predicates that track a special case of cyclicity that occurs in doubly linked lists.

6.1 Instrumentation Predicates that Track Reachability Properties

This section discusses instrumentation predicates that track reachability properties. The instrumentation predicates capture the following properties:

—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 6.1.1 and 6.1.2, respectively.

6.1.1 *Reachability From Individual Program Variables.* Instrumentation predicates that track information about reachability from the individual variables of the program drastically improve the precision of shape analysis, because they keep separate the abstract representations of data structures—and different parts of the same data structure—that are disjoint in the concrete world [Sagiv et al. 1998, p.38]. Therefore, 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 [Jones and Muchnick 1981; 1982; Larus and Hilfinger 1988; Horwitz et al. 1989; Chase et al. 1990; Hendren 1990; Hendren and Nicolau 1990; Landi and Ryder 1991; Stransky 1992; Deutsch 1992; Assmann and Weinhardt 1993; Plevyak et al. 1993; Deutsch 1994; Wang 1994; Sagiv et al. 1998].

The defining formulae for the predicates that track the property "reachable from `x` via 0 or more applications of the field-selector `n`", denoted by $r_{x,n}(v)$, were given in Table IV as equation (5):

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

In order to instantiate the shape-analysis framework with the $r_{x,n}$ predicates, in addition to supplying the definition of $\varphi_{r_{x,n}}(v)$, it is necessary to supply predicate-update formulae, $\varphi^{st}_{r_{x,n}}(v)$ that maintain the correct instrumentation for $r_{x,n}$. Note that in a 3-valued structure S, $\varphi_{r_{x,n}}(v)$ is likely to evaluate to 0 or 1/2 for most individuals. For $[\![\varphi_{r_{x,n}}]\!]^S_3([v \mapsto u])$ to evaluate to 1, there would have to be a path of `n`-edges that all have the value 1, from the individual pointed to by `x` to u. However, the Instrumentation Principle comes into play: As we will see below, in many cases, by maintaining information about cyclicity in addition to reachability, information about the absence of a cycle can be used to update $r_{x,n}$ directly, without reevaluating $\varphi_{r_{x,n}}(v)$.

For programs that use the `List` data-type declaration from Fig. 1(a), predicate-update formulae are listed in Table XII. The predicate-update formula in the `x->n=NULL` case in Table XII, when $z \not\equiv x$, is based on the observation that it is unnecessary to reevaluate $\varphi_{r_{z,n}}(v)$ whenever v does not occur in a directed cycle or is not reachable from `x`.[6]

To be able to identify such situations, we introduce an additional instrumentation predicate, denoted by $c_n(v)$, that records the cyclicity property of v. The defining formula for $c_n(v)$ was given in Table IV as equation (7):

$$\varphi_{c_n}(v) \stackrel{\text{def}}{=} n^+(v, v).$$

As noted in Section 3.1, the cyclicity predicate is useful by itself. It can be used to determine if reference counting would be sufficient for storage management.

For programs that use the `List` data-type declaration from Fig. 1(a), predicate-update formulae for $r_{x,n}(v)$ and $c_n(v)$, are given in Table XII and Table XIII, respectively. Some of these formulae update the value of $r_{x,n}(v)$ in terms of the value of $c_n(v)$, and vice versa.

Let us now consider the predicate-update formulae that appear in Table XII:

—The statement `x = NULL` resets $r_{x,n}$ to 0.

—The statement `x = t` sets $r_{x,n}$ to $r_{t,n}$.

—The statement `x = t->n` sets $r_{x,n}(v)$ to the value of $r_{t,n}(v')$, where v' is an `n`-predecessor of v.

—The statement `x->n = NULL` not only resets the `x`-reachability property $r_{x,n}$, it may also change $r_{z,n}$ when the element directly pointed to by `x` is reached by variable `z`. Furthermore, as illustrated in Fig. 12, in the presence of cycles it is not always obvious how to determine the exact elements whose $r_{z,n}$ properties change. Therefore, the predicate-update formula breaks into two subcases:

  —v appears on a directed cycle and is reachable from the individual pointed to by `x`. In this case, $\varphi_{r_{z,n}}(v)$ is reevaluated (in the structure after the destructive update). For 3-valued structures, this may lead to a loss of precision.

[6] An alternative method for maintaining reachability properties is discussed in Section 7.4.

st	Condition	$\varphi^{st}_{r_{z,n}}(v)$
`x = NULL`	$z \equiv x$	$\mathbf{0}$
	$z \not\equiv x$	$r_{z,n}(v)$
`x = t`	$z \equiv x$	$r_{t,n}(v)$
	$z \not\equiv x$	$r_{z,n}(v)$
`x = t->n`	$z \equiv x$	$\exists v' : r_{t,n}(v') \wedge n(v', v)$
	$z \not\equiv x$	$r_{z,n}(v)$
`x->n = NULL`	$z \equiv x$	$x(v)$
	$z \not\equiv x$	$\begin{cases} \varphi_{r_{z,n}}[n \mapsto \varphi^{st}_n] & \text{if } c_n(v) \wedge r_{x,n}(v) \\ r_{z,n}(v) \wedge \neg(\exists v' : r_{z,n}(v') \wedge x(v') \wedge r_{x,n}(v) \wedge \neg x(v)) & \text{otherwise} \end{cases}$
`x->n = t` (assuming `x->n == NULL`)		$r_{z,n}(v) \vee (\exists v' : r_{z,n}(v') \wedge x(v') \wedge r_{t,n}(v))$
`x = malloc()`	$z \equiv x$	$new(v)$
	$z \not\equiv x$	$r_{z,n}(v) \wedge \neg new(v)$

Table XII. The predicate-update formulae for the instrumentation predicate $r_{z,n}$, for programs that use the `List` data-type declaration from Fig. 1(a). To simplify the presentation, we break the assignment `x->n=t` into two statements: `x->n=NULL`, and `x->n=t` (assuming that `x->n==NULL`).

—v does not appear on a directed cycle or is not reachable from the individual pointed to by `x`. In this case, v fails to be reachable from `z` only if the edge being removed is used on the path from `z` to v.

—After the statement `x = malloc()`, the only element reachable from `x` is the newly allocated element.

Let us now examine the predicate-update formulae that appear in Table XIII:

—The statements `x = NULL`, `x = t`, and `x = t->n`, do not change the store (n-predicates) and thus have no affect on cyclicity.

—If v', the node pointed to by `x`, appears on a cycle, then the statement `x->n = NULL` breaks the cycle involving all the nodes reachable from `x`. (The latter cycle is unique, if it exists). If the node pointed to by `x` does not appear on a cycle, this statement has no affect on cyclicity.

—If `t` reaches the node pointed to by `x`, then the statement `x->n = t` creates a cycle involving all the nodes reachable from `t`. In other cases, no new cycles are created.

—The statement `x = malloc()` sets the cyclicity property of the newly allocated element to 0.

EXAMPLE 6.1. The program shown in Fig. 13 demonstrates the advantage of using reachability predicates for analyzing linked lists.

Fig. 14 shows the structures that occur during the abstract interpretation of `search` with one instrumentation predicate $is(v)$. Fig. 15 shows the details of the application of the abstract transformer in the second iteration. Consider the structure $S_{5.2}$ that occurs in the first iteration: When `y` is advanced down the list by `y = y->n`, $focus_{\{\exists v_1 : y(v_1) \wedge n(v_1, v)\}}$ creates the structure $S_{5.2,f,2}$ in which $u.0$ has been bifurcated into $u.0.1$ and $u.0.0$ (where $[\![\exists v_1 : y(v_1) \wedge n(v_1, v)]\!]^{S_{5.2,f,2}}_3([v \mapsto u.0.1]) = 1$ and $[\![\exists v_1 : y(v_1) \wedge n(v_1, v)]\!]^{S_{5.2,f,2}}_3([v \mapsto u.0.0]) = 0$). The statement transformer

st	$\varphi^{st}_{c_n}(v)$
`x = NULL`	$c_n(v)$
`x = t`	$c_n(v)$
`x = t->n`	$c_n(v)$
`x->n = NULL`	$c_n(v) \wedge \neg(\exists v' : x(v') \wedge c_n(v') \wedge r_{x,n}(v))$
`x->n = t` (assuming `x->n == NULL`)	$c_n(v) \vee \exists v' : x(v') \wedge r_{t,n}(v') \wedge r_{t,n}(v)$
`x = malloc()`	$c_n(v) \wedge \neg new(v)$

Table XIII. The predicate-update formulae for the instrumentation predicate c_n, for programs that use the `List` data-type declaration from Fig. 1(a). To simplify the presentation, we break the assignment `x->n=t` into two statements: `x->n=NULL`, and `x->n=t` (assuming that `x->n==NULL`).

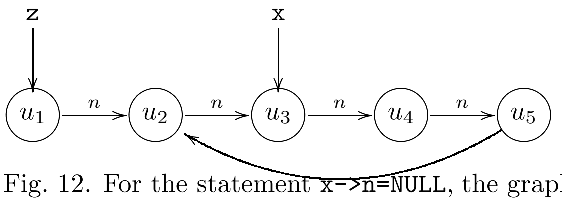


Fig. 12. For the statement `x->n=NULL`, the graph shown above illustrates the chief obstacle for updating reachability information: After the execution of `x->n=NULL`, the elements u_4 and u_5 are no longer reachable from `z`, whereas u_2 (and u_3) are still reachable from `z`. Note that beforehand the value of $r_{z,n}$ is the same for u_2, u_4, and u_5. For such a structure, Table XII reevaluates $\varphi_{r_{z,n}}(v)$.

⟦`y=y->n`⟧ and *coerce* are then applied, which creates the structure $S_{5.2,c,2}$. When $S_{5.2,c,2}$ is converted into a bounded structure, elements $u.1$ and $u.0.0$ are merged into a single element since they have the same values for the unary predicates (i.e., $is = 0, x = 0, y = 0$). Therefore, in $S_{5.2.2}$, this element appears as the single element u. This structure is too conservative: it indicates, for example, that `y` may point to a cyclic structure (which cannot occur in the analyzed program).

In contrast, Fig. 16 shows the structures that occur during the abstract interpretation of `search` with the instrumentation predicates $is(v)$, $r_{x,n}(v)$, and $r_{y,n}(v)$. This version of the analysis does not produce false cycles; that is, the structures that arise represent only acyclic structures.

6.1.2 *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 Table IV as equation (6):

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

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. (With-

```
/* search.c */
#include ‘‘list.h’’
List search(int d, List x) {
  List y;
  assert(acyclic_list(x));
  y = x;
  while (y != NULL && y->data != d) {
    y = y->n;
  }
  return y;
}
```

Fig. 13. A program that searches for an element with a data value `d` in an acyclic singly linked list whose head is pointed to by `x`.

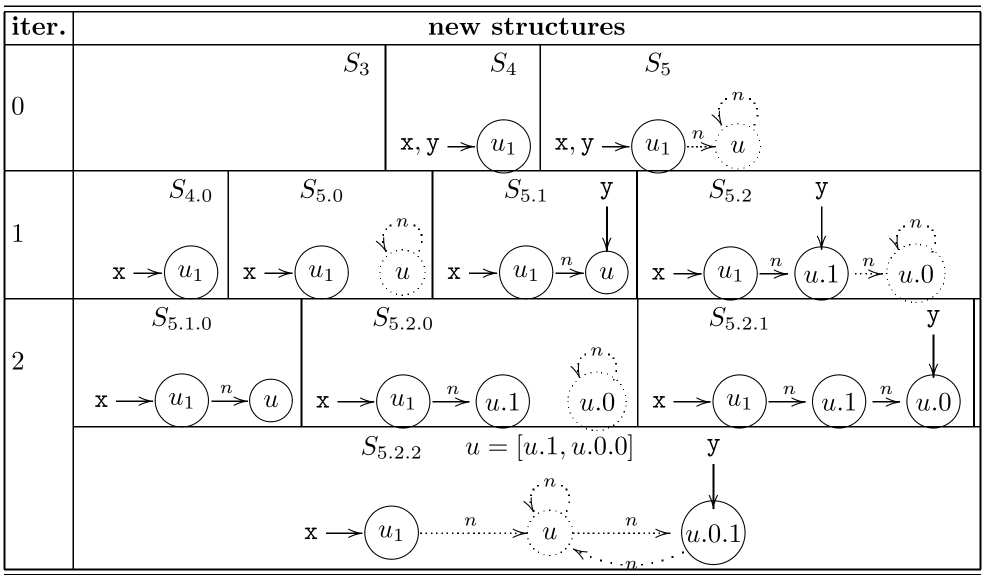


Fig. 14. The structures that occur during the abstract interpretation of `search` with one instrumentation predicate $is(v)$ (See Fig. 15 for a detailed explanation of the application of the abstract transformer to $S_{5.2}$.)

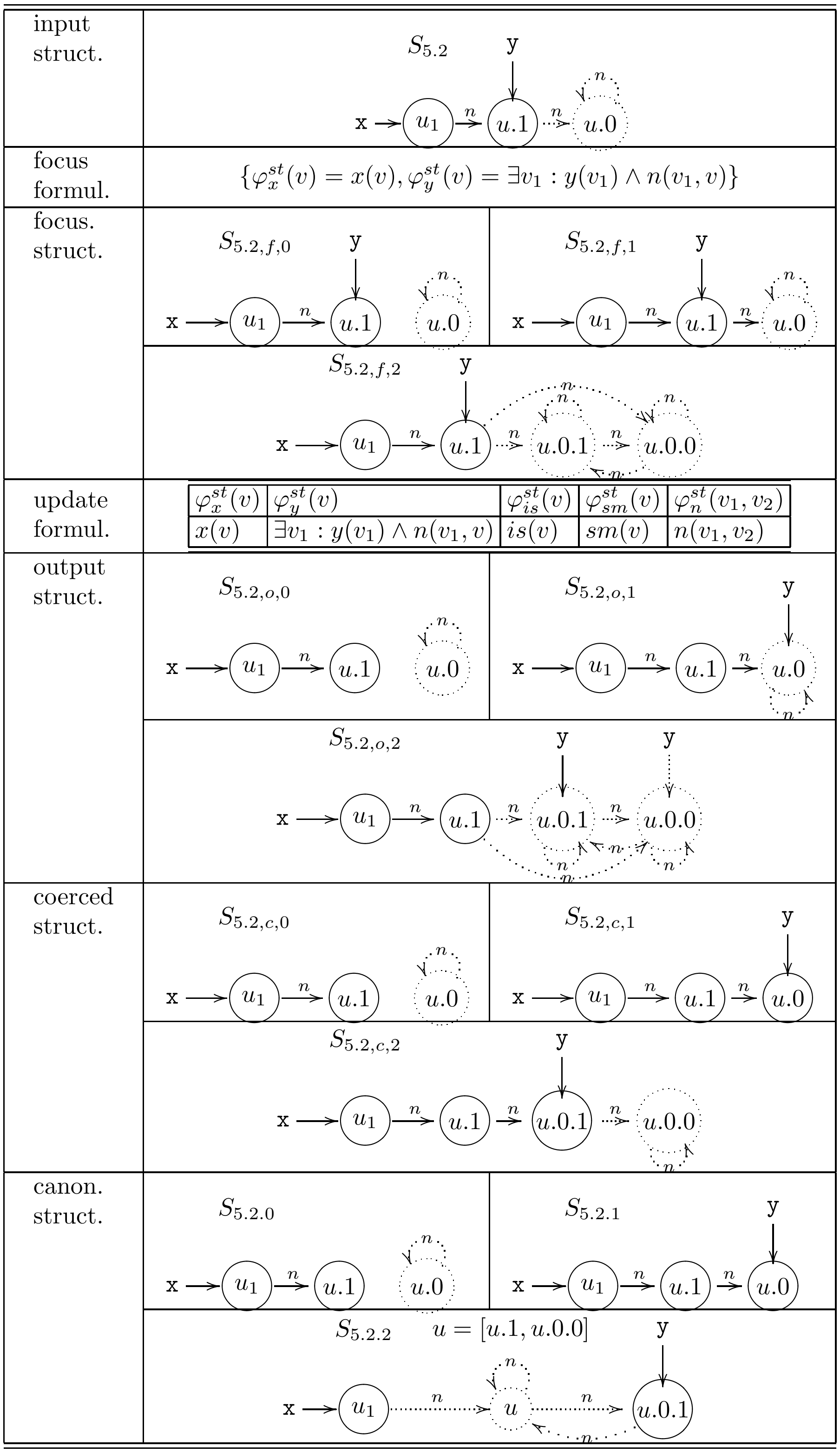


Fig. 15. Detailed explanation of the application of the abstract transformer for the statement `y = y->n` in `search` applied to $S_{5.2}$ shown in Fig. 14.

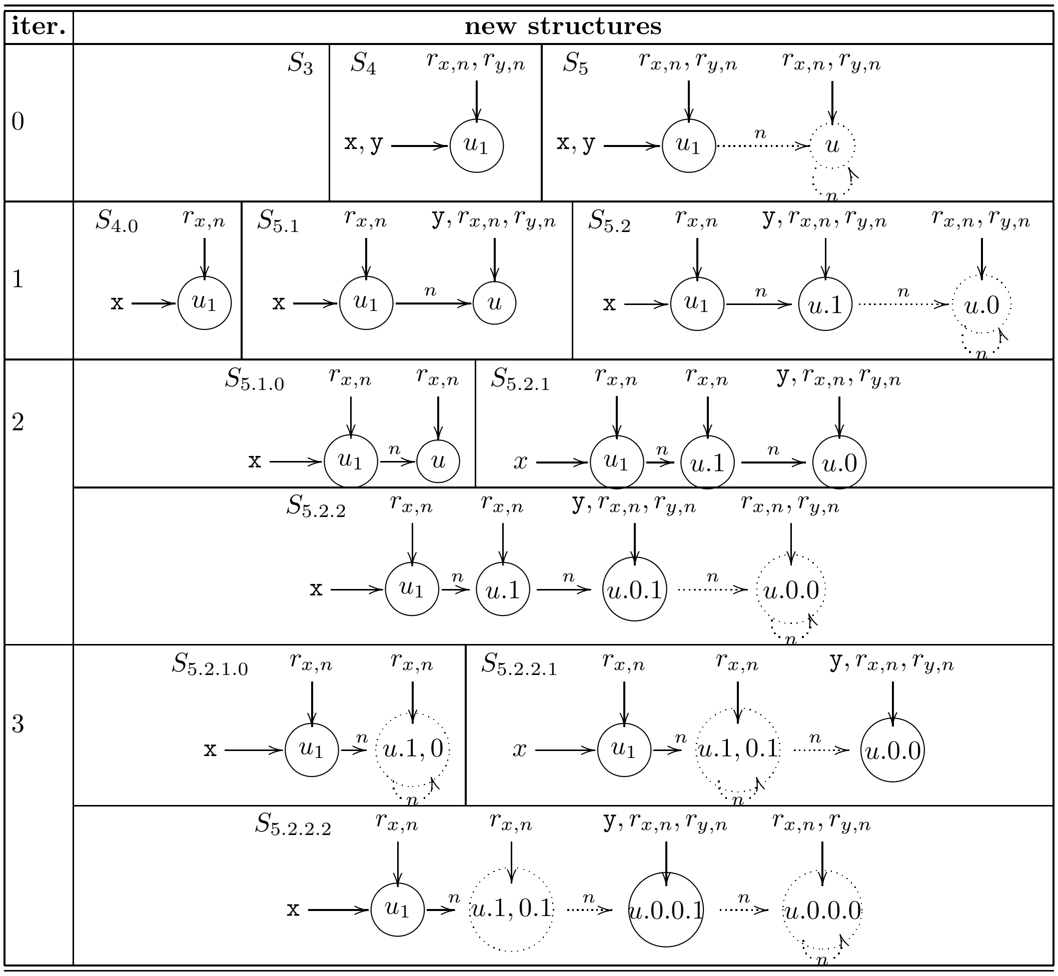


Fig. 16. The structures that occur during the abstract interpretation of `search` with the instrumentation predicates $is(v)$, $r_{x,n}(v)$, and $r_{y,n}(v)$.

```
/* dlist.h */
typedef struct node {
  struct node *f, *b;
  int data;
} *List;
```

Fig. 17. Declaration of a doubly linked-list data type in C.

out 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 Fig. 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 $\varphi_{r_{x,n}}$ 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 $\varphi^{st}_{r_{x,n}}$ predicates:

$$\varphi^{st}_r(v) \stackrel{\text{def}}{=} \bigvee_{\mathtt{x} \in PVar} \varphi^{st}_{r_{x,n}}(v) \tag{65}$$

6.2 Instrumentation Predicates for Doubly Linked Lists

We now briefly sketch the treatment of doubly linked lists. A C declaration of a doubly linked list is given in Fig. 17.

The defining formulae for the predicates $c_{f.b}$ and $c_{b.f}$, which track when forward and backward dereferences "cancel" each other, were given in Table IV as equations (8) and (9):

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

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

The predicate-update formulae for $c_{f.b}$ are given in Table XIV. (The predicate-update formulae for $c_{b.f}$ are not shown because they are dual to those for $c_{f.b}$.)

In addition to $c_{f.b}$ and $c_{b.f}$, we use two different reachability predicates for every variable `z`: (i) $r_{\mathtt{z},f}(v)$, which holds for elements v that are reachable from `z` via 0 or more applications of the field-selector `f`, and (ii) $r_{\mathtt{z},b}(v)$, which holds for elements v that are reachable from `z` via 0 or more applications of the field-selector `b`. Similarly, we use two cyclicity predicates c_f and c_b. The predicate-update formulae for these four predicates are essentially the ones given in Table XII and Table XIII (with n replaced by f and b). (One way in which Table XII should be adjusted is in the case of updating the reachability predicate with respect to one field, say `b`, when the `f`-field is traversed, i.e., via `x = t->f`. In this case, the predicates $c_{f.b}$ and $c_{b.f}$ can be used to avoid an over-conservative solution [Lev-Ami 2000]).

We have already demonstrated how the shape-analysis algorithm works as a pointer is advanced along a singly linked-list, as in the body of `search` (see Fig. 16). The shape-analysis algorithm works in a similar fashion when a pointer is advanced

st	$\varphi^{st}_{c_{f.b}}(v)$
`x = NULL`	$c_{f.b}(v)$
`x = t`	$c_{f.b}(v)$
`x = t->f`	$c_{f.b}(v)$
`x->f = NULL`	$c_{f.b}(v) \lor x(v)$
`x->b = NULL`	$c_{f.b}(v) \land \neg\exists v_1 : x(v_1) \land b(v_1, v)$
`x->f = t` (assuming `x->f == NULL`)	$\begin{cases} \forall v_1 : t(v_1) \Rightarrow b(v_1, v) & \text{if } x(v) \\ c_{f.b}(v) & \text{otherwise} \end{cases}$
`x->b = t` (assuming `x->b == NULL`)	$c_{f.b}(v) \lor (t(v) \land \exists v_1 : f(v, v_1) \land x(v_1))$
`x = malloc()`	$c_{f.b}(v) \lor new(v)$

Table XIV. The predicate-update formulae for the instrumentation predicate $c_{f.b}$.

along a doubly linked-list. Therefore, in this section, we consider the operation `splice`, shown in Fig. 18, that splices an element with a data value `d` into a doubly linked list after an element pointed to by `p`. (We will assume that this operation occurs after a search down the list has been carried out, and that the variable that points to the head of the list is named `l`.)

Fig. 19 illustrates the abstract interpretation of `splice` under the following conditions: `p` points to some element in the list beyond the second element, and the tail of `p` is not NULL. (This is the most interesting case since it exhibits all of the possible indefinite edges arising in a call on `splice`.) Preceding row by row in Fig. 19, we observe the following changes:

—In the initial structure, the values of $c_{f.b}$ and $c_{b.f}$ are 1 for all elements, since in all of the list elements forward and backward dereferences cancel.

—Immediately after a new heap-cell is allocated and its address assigned to `e`, $c_{f.b}$ and $c_{b.f}$ are both trivially true for the new element since this element's `f` and `b` components do not point to any element. Note that in the last row of Table XIV, the value of $c_{f.b}$ for a newly allocated element's is set to 1.

—The assignment to the `data` field of `e` does not change the structure. The assignment `t = p->f` materializes a new element whose $c_{f.b}$ and $c_{b.f}$ predicate values are 1:
In the second structure shown in Fig. 19, the values of $c_{f.b}$ and $c_{b.f}$ are both 1 for the summary element pointed to by `p->f`. In going from the second structure to the third structure, these predicate values are not changed by the predicate-update formula in the `x = t->f` row of Table XIV.

—The assignment `e->f = t`, is performed in two stages: (i) `e->f = NULL` and then (ii) `e->f = t` assuming that `e->f == NULL`. The first stage has no effect since the value of $c_{f.b}$ is 1 for the element pointed to by `e`. But then `e->f = t` changes the value of $c_{f.b}$ to 0 for the element pointed to by `e`, and changes the values of $r_{e,f}$ and $r_{e,b}$ to 1 for the elements transitively pointed to by `t`.
The fourth structure shown in Fig. 19 is produced by the `x->f = t` row of Table XIV, for `x` $\equiv$ `e`, since $\forall v_1 : t(v_1) \Rightarrow b(v_1, v)$ evaluates to 0 for the element v pointed to by `e`. Also, by the `x->n = t` row of Table XII, for `x` $\equiv$ `e` and $n = f$,

```
/* splice.c */
#include ‘‘dlist.h’’
void splice(int v, DList p) {
  DList e, t;
  e = (DList)malloc(sizeof(struct DListNode));
  e->data = v;
  t = p->f;
  e->f = t;
  if (t != NULL)
     t->b = e;
  p->f = e;
  e->b = p;
}
```

Fig. 18. A program that splices an element with a data value `d` into a doubly linked list after an element pointed to by `p`. We will assume that the variable that points to the head of the doubly linked list is named `l`.

the elements reachable in the forward direction from `t` acquire the value 1 for $r_{e,f}$.

—The assignment `t->b = e`, is performed in two stages: (i) `t->b = NULL` and then (ii) `t->b = NULL`, assuming that `t->b == NULL`.
The assignment `t->b = NULL` changes the value of $c_{f.b}$ to 0 for the element pointed to by `p`. The fifth structure shown in Fig. 19 is produced by the `x->b = NULL` row of Table XIV, for `x` $\equiv$ `t` and `t` $\equiv$ `e`, since $\neg\exists v_1 : t(v_1) \wedge b(v_1, v)$ evaluates to 0 for the element pointed to by `p`. In the second stage, the assignment `t->b = e` changes the value of $c_{f.b}$ to 1 for the element pointed to by `e`: By the `x->b = t` row of Table XIV, for `x` $\equiv$ `t` and `t` $\equiv$ `e`, the formula $e(v) \wedge \exists v_1 : f(v, v_1) \wedge t(v_1)$ evaluates to 1 for the element pointed to by `e`.
Also, by the `x->n = NULL` row of Table XII, for `x` $\equiv$ `t` and $n \equiv b$, the elements reachable in the backward direction from `t` are no longer reachable from `t`. Finally, by the `x->n = t` row of Table XII, for `x` $\equiv$ `t`, `t` $\equiv$ `e`, and $n \equiv b$, the nodes reachable in the backward direction from `e` are now reachable from `t`.

—The assignment `p->f = e` involves an implicit assignment `p->f=NULL`. This causes the elements reachable from `p->f` to no longer be reachable from `p` and `l`. However, the assignment `p->f = e` restores the reachability properties $r_{p,f}$ and $r_{l,f}$ to all the elements reachable from `e` along the forward direction.

—The assignment `e->b = p` restores the reachability properties $r_{e,b}$ and $r_{t,b}$ to all elements reachable from `p` along the backward direction and $c_{f.b}$ for the element pointed to by `p`.

7. RELATED WORK

This paper presents results from an effort to clarify and extend our previous work on shape analysis [Sagiv et al. 1998]. Compared with [Sagiv et al. 1998], the major differences are

—A single specific shape-analysis algorithm was presented in [Sagiv et al. 1998].

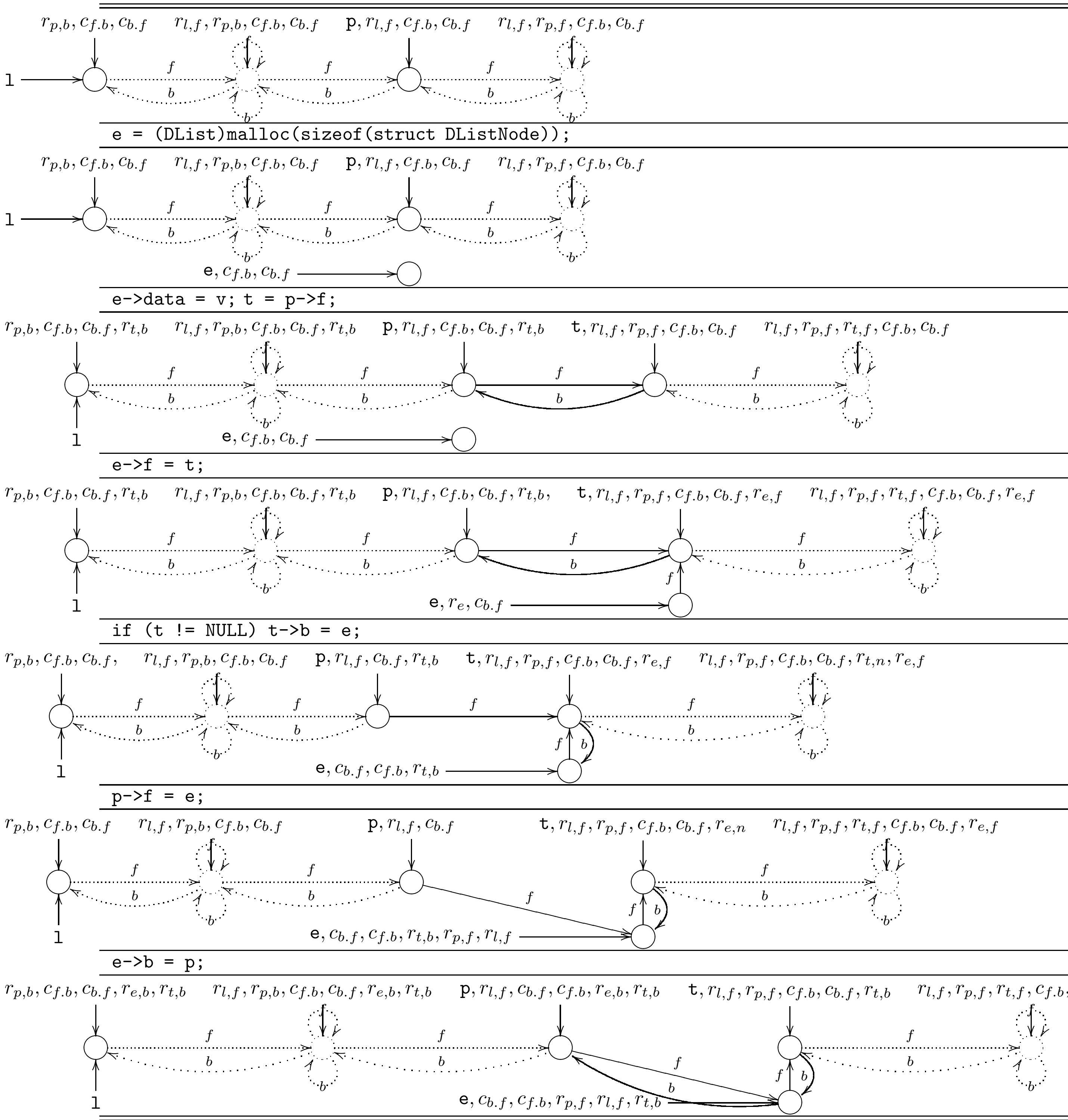


Fig. 19. The abstract interpretation of the `splice` procedure applied to a doubly linked list whose head is pointed to `l`. Variable `p` points to some element in the list beyond the second element, and the tail of `p` is assumed to be non-NULL. For brevity, $r_{z,f}(v)$ and $r_{z,b}(v)$ are not shown when v is directly pointed to by `z`.

The present paper presents a *parametric* framework for shape analysis: It provides the basis for generating different shape-analysis algorithms by varying the instrumentation predicates used.

—This paper uses different instantiations of the parametric framework to show how shape analysis can be performed for a variety of different kinds of linked data structures.

—The shape-analysis algorithm in [Sagiv et al. 1998] was cast as an abstract interpretation, in which the abstract transfer functions transformed shape graphs to shape graphs. The present paper is based on logic, and "shape graphs" are replaced by "3-valued logical structures". The use of logic has many advantages. The most important of these is that it relieves the designer of a particular shape analysis from many of the burdensome tasks that the methodology of abstract interpretation ordinarily imposes. In particular, (i) the abstract semantics falls out automatically from the concrete semantics, and (ii) there is no need for a proof that a particular instantiation of the shape-analysis framework is correct—the soundness of *all* instantiations of the framework follows from a single meta-theorem, the Embedding Theorem, which shows that information extracted from a 3-valued structure is sound with respect to information extracted from a corresponding 2-valued structure.

A substantial amount of material covering previous work on pointer analysis, alias analysis, and shape analysis is presented in [Sagiv et al. 1998]. In the remainder of this section, we confine ourselves to the work most relevant to the present paper.

7.1 Previous Work on Shape Ansalysis

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

—The algorithms of [Wang 1994; Sagiv et al. 1998] map unbounded-size stores into bounded-size abstractions by collapsing concrete cells that are not directly pointed to by program variables into one abstract cell, whereas concrete cells that are pointed to by different sets of variables are kept apart in different abstract cells. As discussed in Section 4.4, these algorithms are captured in the framework by using abstraction predicates of the form pointed-to-by-variable-`x` (for all $x \in PVar$).

—The algorithm of [Jones and Muchnick 1981], which collapses individuals that are not reachable from a pointer variable in k or fewer steps, for some fixed k, can be captured in our framework by using instrumentation predicates of the form "reachable-from-`x`-via-access-path-α", for $|\alpha| \leq k$.

—The algorithms of [Jones and Muchnick 1982; Chase et al. 1990] can be captured in the framework by introducing unary core predicates that record the allocation sites of heap cells.

—The algorithm of [Plevyak et al. 1993] can be captured in the framework using the predicates $c_{f.b}(v)$ and $c_{b.f}(v)$ (see Table III and Table IV).

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

Roughly speaking, the chief alternative to the use of shape graphs involves representing may-aliases between pointer-access paths [Hendren and Nicolau 1990; Landi and Ryder 1991; Deutsch 1992; 1994; Sagiv et al. 1998]. Compared with shape graphs, these methods have certain drawbacks. In particular, shape graphs represent the topological properties of the store directly, which allows certain operations, such as destructive updates, to be tracked more precisely. In addition, shape graphs are a more intuitive mechanism for reporting information back to a human, and thus may be more useful in program-understanding tools. On the other hand, representations of may-aliases can be more compact than shape graphs, and some may-alias algorithms are capable of representing information that goes beyond the capabilities of bounded structures [Deutsch 1992; 1994].

7.2 The Use of Logic for Pointer Analysis

Jensen et al. defined a decidable logic for describing properties of linked data structures, and showed how it could be used to verify properties of programs written in a subset of Pascal [Jensen et al. 1997]. Because of the methods on which their decision procedure is based, the logic is limited to the case of linked lists in which no sharing of common tails occurs. (Known theoretical results imply that the work can be extended to handle programs that manipulate unshared trees.) The method that has been described in the present paper can handle some programs that manipulate shared structures (as well as some circular structures, such as doubly linked lists).

Benedikt et al. defined a decidable logic, called L_r, for describing properties of linked data structures [Benedikt et al. 1999]. They showed how a generalization of Hendren's path-matrix descriptors [Hendren 1990; Hendren and Nicolau 1990] can be represented by L_r formulae, as well as how the variant of static shape graphs defined by Sagiv et al. [Sagiv et al. 1998] can be represented by L_r formulae. This correspondence provides insight into the expressive power of path matrices and static shape graphs. It also has interesting consequences for extracting information from the results of program analyses, in that it provides a way to *amplify* the results obtained from known analyses:

—By translating the structure descriptors obtained from the techniques given in [Hendren 1990; Hendren and Nicolau 1990; Sagiv et al. 1998] to L_r formulae, it is possible to determine if there is any store at all that corresponds to a given structure descriptor. This makes it possible to determine whether a given structure descriptor contains any useful information.

—Decidability provides a mechanism for reading out information obtained by existing shape-analysis algorithms, without any additional loss of precision over that inherent in the shape-analysis algorithm itself.

The 3-valued structures used in this paper are more general than L_r; that is, not all properties that we are able to capture using 3-valued structures can be expressed in L_r. Thus, it is not clear to us whether L_r (or a decidable extension of L_r) can be used to amplify the results obtained via the techniques described in the present paper.

Other formalisms for describing linked data structures include ADDS [Hendren et al. 1992] and graph grammars [Fradet and Metayer 1997]. Like L_r, these formalisms are annotation languages for expressing loop invariants and pre- and post-conditions of statements and procedures. It should be noted that L_r, ADDS, and the graph grammars of Fradet and Le Métayer are, in their present stage of development, mainly useful as a documentation notation for type definitions, function arguments, and function return values. In contrast, a decision precedure for Hoare triples over loop-free code (without arithmetic) is known for the store logic of [Jensen et al. 1997]. Consequently, the work of [Jensen et al. 1997] can be used for verifying programs that contain loops when each loop is annotated with a loop invariant. The instantiations of our parametric shape-analysis framework address yet another problem—that of inferring shape annotations from a program automatically. In the latter problem, the analysis algorithm is provided with (at most) a description of a procedure's inputs.

Morris studied the use of a reachability predicate "$x \to v \mid K$" for establishing properties of programs that manipulate linked lists and trees [Morris 1982]. The predicate $x \to v \mid K$ means "v is a node reachable from variable x via a path that avoids nodes pointed to by variables in set K". Morris discussed techniques that, given a statement and a post-condition, generate a formula that captures the weakest-precondition. It is not clear to us how this relates to our predicate-update formulae, which update the values of predicates after the execution of a pointer-manipulation statement.

7.3 The Embedding Theorem

Despite the naturalness and simplicity of the Embedding Theorem, this theorem appears not to have been known previously [Kunen 1998; Lifschitz 1998]. The closest concept that we have found in the literature is the notion of embedding discussed by Bell and Machover [Bell and Machover 1977, page 165]. For them, an embedding of one 2-valued structure into another is a one-to-one, truth-preserving mapping. However, this notion is unsuitable for abstract interpretation of programs that manipulate heap-allocated storage, because in abstract interpretation it is necessary to have a way to associate the structures that arise in the concrete semantics, which are of *arbitrary size*, with abstract structures of some *fixed size*. To accomplish this, we have introduced "truth-blurring" onto mappings (e.g., tight embeddings). The Embedding Theorem ensures that the meaning of a formula in the "blurred" (abstract) world is consistent with the formula's meaning in the original (concrete) world.

7.4 Relationship to First-Order Incremental Evaluation Systems

The issue of devising predicate-update formulae for instrumentation predicates, discussed in Sections 4, 5, and 6, is related to previous work on "first-order incremental evaluation schemes" (FOIES) [Dong and Su 1995; 1998] and on "dynamic descriptive complexity" [Patnaik and Immerman 1997; Immerman 1999]. These papers address the problem of maintaining one or more "auxiliary" predicates after new tuples are inserted into or deleted from the base predicates of a logical structure. For a variety of different problems, they present appropriate predicate-update formulae. In our work, the evaluation of a given statement st in the program adds tuples to, or deletes tuples from, the core predicates (in each of the first-order structures that arise just before st). The predicate-update formulae serve to generate the updated values of the instrumentation predicates.

Section 6.1 discusses predicate-update formulae that maintain reachability and cyclicity instrumentation predicates. In the case of shape analysis of programs that manipulate data structures of type `List`, the predicate-update problem is closely related to the work by Dong and Su on maintaining reachability properties in "1-path graphs"—graphs in which two nodes s and t are connected by at most one path [Dong and Su 1995; 1998]. In the case of `List`-manipulation programs, because each individual can have at most one outgoing n-edge, all 2-valued structures of the concrete semantics are necessarily 1-path graphs.[7]

We were unable to adopt the Dong-Su solution to the reachability-maintenance problem for 1-path graphs given in [Dong and Su 1995]. It involves the use of an additional auxiliary predicate, $DEP(x, \alpha, \beta, y)$, which records, for each pair of individuals x and y, whether x reaches y along an acyclic path that uses edge $\langle \alpha, \beta \rangle$. Unfortunately, when a tight embedding is performed on a structure augmented with the DEP predicate, much of the DEP information goes to 1/2 (e.g., when one or more of x, α, β, and y are summary nodes). Because the DEP predicate is updated in terms of the DEP predicate [Dong and Su 1995], many tuples stored in DEP (as well as the reachability information derived from DEP) rapidly acquire the value 1/2.

In contrast, the instrumentation predicates adopted in Section 6.1 track reachability information with respect to individuals that are pointed to by program variables: Because the core predicates include the pointed-to-by-variable-`x` predicates for all $x \in PVar$, such individuals are never abstracted to summary nodes, and consequently accurate reachability information can be maintained in the 3-valued structures of the abstract semantics. There is one case in which we throw up our hands and re-evaluate reachability directly; however, this only involves nodes that are part of (or were once part of) a cycle.

8. CONCLUSIONS

We conclude with a few general observations about the material that has been developed in the paper.

8.1 The Advantages of Using Logic

With the methodology of abstract interpretation, it is not always an easy task to obtain an appropriate abstract semantics; abstract-interpretation papers often contain exhaustive (and exhausting) proofs to show the soundness of a given abstract semantics with respect to a given concrete semantics. With the approach

[7] For programs that manipulate `List` structures, the 2-valued structures of the concrete semantics also fall into the smaller class of "deterministic graphs" mentioned by Patnaik and Immerman [Patnaik and Immerman 1997; Immerman 1999].

taken in this paper, however, this is not the case: The abstract semantics falls out automatically from the concrete semantics; soundness follows from a single meta-theorem (the Embedding Theorem), which shows that information extracted from a 3-valued structure is sound with respect to information extracted from a corresponding 2-valued structure.

Because it is not generally true of abstract interpretation that the abstract semantics falls out automatically from the concrete semantics, it is worthwhile to reiterate the principle on which our work is based:

OBSERVATION 8.1. [**Reinterpretation Principle**]. *The advantage of using Kleene's 3-valued 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 a statement about the 2-valued world or the 3-valued world. The consistency of these two views is ensured by the Embedding Theorem.*

3-valued logic also provides a way to tune the amount of space used by a shape-analysis algorithm: If too many different structures arise at a given program point, one can always reduce the number of structures by promoting definite values of tuples to 1/2, applying *t_embed*, and retaining only maximal structures, until the number of structures falls to the desired number.

8.2 Propagation of Formulae Versus Propagation of Structures

It is interesting to compare the machinery developed in this paper with the approach taken in methodologies for program development based on weakest preconditions [Dijkstra 1976; Gries 1981], and also in systems for automatic program verification [King 1969; Deutsch 1973; Constable et al. 1982], where assertions (formulae) are pushed backwards through statements. The justification for propagating information in the backwards direction is that it avoids the existential quantifiers that arise when assertions are pushed in the forwards direction to generate strongest postconditions. Ordinarily, strongest postconditions present difficulties because quantifiers accumulate, forcing one to work with larger and larger formulae.

In the shape-analysis framework developed in this paper, an abstract shape transformer can be viewed as computing a safe approximation to a statement's strongest post-condition: The application of an abstract statement transformer to a 3-valued logical structure describing a set of stores S that arise before a given statement st creates a set of 3-valued logical structures that covers all of the stores that could arise from applying st to members of S. However, the shape-analysis framework works at the semantic level—that is, it operates directly on explicit representations of logical structures, rather than on an implicit representation, such as a logical formula.[8] It is true that new abstract heap-cells are materialized when necessary via the Focus operation; however, because the fixed-point-finding algorithm keeps performing abstraction (via *t_embed*), 3-valued logical structures cannot grow to be of unbounded size.

[8]However, see [Benedikt et al. 1999] for a discussion of how shape graphs can be converted to logical formulae.

8.3 Biased Versus Unbiased Static Program Analysis

Many of the classical dataflow-analysis algorithms use bit vectors to represent the characteristic functions of set-valued dataflow values. This corresponds to a logical interpretation (in the abstract semantics) that uses two values. It is *definite* on one of the bit 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 true/may be false". Many other static-analysis algorithms have a similar character.

Conventional wisdom holds that static analysis must inherently have such a one-sided bias. However, the material developed in this paper shows that while *indefiniteness* is inherent (i.e., a static analysis is unable, in general, to give a definite answer), one-sidedness is not: By basing the abstract semantics on 3-valued logic, definite truth and definite falseness can both be tracked, with the third value, 1/2, capturing indefiniteness.

This outlook provides some insight into the true nature of the values that arise in other work on static analysis:

—A one-sided analysis that is precise with respect to "false" and conservative with respect to "true" is really a 3-valued analysis over 0, 1, and 1/2 that conflates 1 and 1/2 (and uses "true" in place of 1/2).

—Likewise, an analysis that is precise with respect to "true" and conservative with respect to "false" is really a 3-valued analysis over 0, 1, and 1/2 that conflates 0 and 1/2 (and uses "false" in place of 1/2).

In contrast, the analyses developed in this paper are unbiased: They are precise with respect to both 0 and 1, and use 1/2 to capture indefiniteness.

Acknowledgements

We are grateful to T. Lev-Ami for his implementation and his suggested improvements to the precision of the algorithms presented in this paper. We are also grateful for the helpful comments of A. Avron, T. Ball, M. Benedikt, N. Dor, M. Fahndrich, M. Gitik, K. Kunen, V. Lifschitz, F. Nielson, H.R. Nielson, M. O'Donnell, A. Rabinovich, and K. Sieber. We thank K.H. Rose for the XY LaTeX package.

REFERENCES

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

Bell, J. and Machover, M. 1977. *A Course in Mathematical Logic*. North-Holland Publishing Co.

Benedikt, M., Reps, T., and Sagiv, M. 1999. A decidable logic for describing linked data structures. In *Proceedings of the 1999 European Symposium On Programming*. 2–19.

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

Constable, R., Johnson, S., and Eichenlaub, C. 1982. *Introduction to the PL/CV2 Programming Logic*. Lec. Notes in Comp. Sci., vol. 135. Springer-Verlag.

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

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

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

Deutsch, L. 1973. An interactive program verifier. Ph.D. thesis, Univ. of California, Berkeley, CA.

Dijkstra, E. 1976. *A Discipline of Programming.* Prentice-Hall.

Dong, G. and Su, J. 1995. Incremental and decremental evaluation of transitive closure by first-order queries. *Inf. & Comput. 120*, 101–106.

Dong, G. and Su, J. 1998. Arity bounds in first-order incremental evaluation and definition of polynomial time database queries. *Journal of Computer and System Sciences 57,* 3 (Dec.), 289–308.

Evans, D. 1996. Static detection of dynamic memory errors. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.* Available at "http://larch-www.lcs.mit.edu:8001/∼ evs/pldi96-abstract.html".

Fradet, P. and Metayer, D. L. 1997. Shape types. In *Symp. on Princ. of Prog. Lang.* ACM Press, New York, NY.

Ginsberg, M. 1988. Multivalued logics: A uniform approach to inference in artificial intelligence. *Comp. Intell. 4*, 265–316.

Gries, D. 1981. *The Science of Programming.* Springer-Verlag.

Hendren, L. 1990. Parallelizing programs with recursive data structures. Ph.D. thesis, Cornell Univ., Ithaca, NY.

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

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

Hoare, C. 1975. Recursive data structures. *Int. J. of Comp. and Inf. Sci. 4,* 2, 105–132.

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

Immerman, N. 1999. *Descriptive Complexity.* Springer-Verlag.

Jensen, J., Joergensen, M., N.Klarlund, and Schwartzbach., M. 1997. Automatic verification of pointer programs using monadic second-order logic. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*

Jones, N. and Muchnick, S. 1981. Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, NJ, Chapter 4, 102–131.

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

King, J. 1969. A program verifier. Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, PA.

Kleene, S. 1987. *Introduction to Metamathematics*, Second ed. North-Holland.

Kunen, K. 1998. Personal communication.

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

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

Lev-Ami, T. 2000. TVLA: A framework for Kleene based static analysis. M.S. thesis, Tel-Aviv University. Available at http://www.math.tau.ac.il/∼tla.

Lev-Ami, T., Reps, T., Sagiv, M., and Wilhelm, R. 2000. Putting static analysis to work for verification: A case study. Submitted for publication.

Lifschitz, V. 1998. Personal communication.

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

MORRIS, J. 1982. Assignment and linked data structures. In *Theoretical Foundations of Programming Methodology*, M. Broy and G. Schmidt, Eds. D. Reidel Publishing Co., Boston, MA, 35–41.

NIELSON, F., NIELSON, H., AND SAGIV, M. 1999. A Kleene analysis of mobile ambients. In *Proceedings of the 2000 European Symposium On Programming.*

PATNAIK, S. AND IMMERMAN, N. 1997. Dyn-FO: A parallel, dynamic complexity class. *Journal of Computer and System Sciences 55,* 2 (Oct.), 199–209.

PLEVYAK, J., CHIEN, A., AND KARAMCHETI, V. 1993. Analysis of dynamic structures for efficient parallel execution. In *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Lec. Notes in Comp. Sci., vol. 768. Springer-Verlag, Portland, OR, 37–57.

SAGIV, M., REPS, T., AND WILHELM, R. 1996. Solving shape-analysis problems in languages with destructive updating. In *Symp. on Princ. of Prog. Lang.* ACM Press, New York, NY.

SAGIV, M., REPS, T., AND WILHELM, R. 1998. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst. 20,* 1 (Jan.), 1–50.

SAGIV, M., REPS, T., AND WILHELM, R. 1999. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.* Available at "http://www.cs.wisc.edu/wpis/papers/popl99.ps".

SAGIV, S., FRANCEZ, N., RODEH, M., AND WILHELM, R. 1998. A logic-based approach to data flow analysis problems. *Acta Inf. 35,* 6 (June), 457–504.

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

WANG, E. Y.-B. 1994. Analysis of recursive types in an imperative language. Ph.D. thesis, Univ. of Calif., Berkeley, CA.

A. PROOF OF THE EMBEDDING THEOREM

Theorem 3.11 *Let $S = \langle U^S, \iota^S \rangle$ and $S' = \langle U^{S'}, \iota^{S'} \rangle$ be two structures, and let $f\colon U^S \to U^{S'}$ be a function such that $S \sqsubseteq^f S'$. Then, for every formula φ and complete assignment Z for φ, $[\![\varphi]\!]_3^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S'}(f \circ Z)$.*

Proof: By the De Morgan laws it is sufficient to show the theorem for formulae involving $\wedge$, $\neg$, $\exists$, and TC. The proof is 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

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

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

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

Let us now show that

$$[\![v_1 = v_2]\!]_3^S(Z) \sqsubseteq [\![v_1 = v_2]\!]_3^{S'}(f \circ Z).$$

First, if $[\![v_1 = v_2]\!]_3^{S'}(f \circ Z) = 1/2$ then the theorem holds for $v_1 = v_2$, trivially. Second, if $[\![v_1 = v_2]\!]_3^{S'}(f \circ Z) = 1$ 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, $Z(v_1) = Z(v_2)$ and

$\iota^S(sm)(Z(v_1)) = 0$ both hold. Hence, by Definition 3.4, $[\![v_1 = v_2]\!]_3^S(Z) = 1$. Finally, suppose that $[\![v_1 = v_2]\!]_3^{S'}(f \circ Z) = 0$ holds. In this case, by Definition 3.4, $f(Z(v_1)) \neq f(Z(v_2))$. Therefore, $Z(v_1) \neq Z(v_2)$, and by Definition 3.4 $[\![v_1 = v_2]\!]_3^S(Z) = 0$.

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 $[\![\varphi]\!]_3^{S'}(Z) = 1/2$, then the theorem holds trivially. Therefore assume that $[\![\varphi]\!]_3^{S'}(f \circ Z) \in \{0, 1\}$. We distinguish between the following cases:

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

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

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

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

Case 1: $[\![\neg\varphi_1]\!]_3^{S'}(f \circ Z) = 0$.
In this case, $[\![\varphi_1]\!]_3^{S'}(f \circ Z) = 1$.
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: $[\![\neg\varphi_1]\!]_3^{S'}(f \circ Z) = 1$.
In this case, $[\![\varphi_1]\!]_3^{S'}(f \circ Z) = 0$.
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. $\varphi \equiv \exists v_0 : \varphi_1$. The proof splits into the following subcases:

Case 1: $[\![\exists v_1 : \varphi_1]\!]_3^{S'}(f \circ Z) = 0$.
In this case, for all $u \in U^S$, $[\![\varphi_1]\!]_3^{S'}((f \circ Z)[v_1 \mapsto f(u)]) = 0$. 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:$[\![\exists v_1 : \varphi_1]\!]_3^{S'}(f \circ Z) = 1$.
In this case, there exists a $u' \in U^{S'}$ such that $[\![\varphi_1]\!]_3^{S'}((f \circ Z)[v_1 \mapsto u']) = 1$. Because f is surjective, there exists a $u \in U^S$ such that $f(u) = u'$ and $[\![\varphi_1]\!]_3^{S'}((f \circ Z)[v_1 \mapsto f(u)]) = 1$. 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. $\varphi \equiv (TC\ v_1, v_2 : \varphi_1)(v_3, v_4)$. The proof splits into the following subcases:

Case 1: $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S'}(f \circ Z) = 1$.
By Definition 3.4, there exist $u'_1, u'_2, \ldots, u'_{n+1} \in U^{S'}$ such that for all $1 \leq i \leq n$, $[\![\varphi_1]\!]_3^{S'}((f \circ Z)[v_1 \mapsto u'_i, v_2 \mapsto u'_{i+1}]) = 1$, $(f \circ Z)(v_3) = u'_1$, and $(f \circ Z)(v_4) = u'_{n+1}$. 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, $Z(v_3) = u_1$, $Z(v_4) = u_{n+1}$, 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: $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S'}(f \circ Z) = 0$.
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 $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S'}(f \circ Z) = 0$, 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 $Z(v_3) = u_1$, $Z(v_4) = u_{n+1}$, 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 $(f \circ Z)(v_3) = u'_1$, and $(f \circ Z)(v_4) = u'_{n+1}$ and for all $1 \leq i \leq n$, $[\![\varphi_1]\!]_3^{S'}((f \circ Z)[v_1 \mapsto u'_i, v_2 \mapsto u'_{i+1}]) \neq 0$. Therefore, by Definition 3.4, $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S'}(f \circ Z) \neq 0$, which is a contradiction.

B. OTHER PROOFS

B.1 Properties of the Generated 3-Valued Constraints

Lemma 5.13 *For every pair of structures $S^\natural \in$ 2-CSTRUCT$[\mathcal{P}, F]$ and $S \in$ 3-STRUCT$[\mathcal{P}]$ such that S is a tight embedding of $S^\natural$, $S \models \widehat{r}(F)$.*

Proof: Let $S^\natural \in$ 2-CSTRUCT$[\mathcal{P}, F]$ and $S \in$ 3-STRUCT$[\mathcal{P}]$ be a pair of structures such that S is a tight embedding of $S^\natural$ via function $f\colon U^{S^\natural} \to U^S$. We need to show that $S \models \widehat{r}(F)$.

Let $\varphi' \in F$ and let us show that $S \models r(\varphi')$. If $\varphi' \equiv \forall v_1, v_2, \ldots, v_k : \varphi$, then, since $S^\natural \models \varphi'$, for all assignments $Z^\natural$ for $v_1, v_2, \ldots, v_k$ drawn from $U^{S^\natural}$, $[\![\varphi]\!]_3^{S^\natural}(Z^\natural) = 1$. Therefore, by the Embedding Theorem $[\![\varphi]\!]_3^S(f \circ Z^\natural) \neq 0$. But since f is surjective we conclude that for all assignments Z for $v_1, v_2, \ldots, v_k$ drawn from U^S, $[\![\varphi]\!]_3^S(Z) \neq 0$, and therefore $S \models r(\varphi')$.

Let us now show that $S \models r(\varphi')$ for $\varphi' \equiv \forall v_1, v_2, \ldots, v_k : \varphi \Rightarrow a^b$, where a is an atomic formula, $a \not\equiv sm(v)$, and $b \in \{0, 1\}$. Let Z be an assignment for $v_1, v_2, \ldots, v_k$ drawn from U^S. If $[\![\varphi]\!]_3^S(Z) \neq 1$, then by definition $S, Z \models \varphi \rhd a^b$. Therefore, assume that $[\![\varphi]\!]_3^S(Z) = 1$ and let us show that $[\![a^b]\!]_3^S(Z) = 1$. Note that for every assignment $Z^\natural$ such that $f \circ Z^\natural = Z$, $[\![\varphi]\!]_3^S(Z) = 1$ implies, by the Embedding Theorem, that $[\![\varphi]\!]_3^{S^\natural}(Z^\natural) = 1$. Therefore, because $S^\natural \models \varphi'$, we have

$$[\![a^b]\!]_3^{S^\natural}(Z^\natural) = 1. \tag{66}$$

The remainder of the proof splits into the following cases:

Case 1: $b = 1$ and $a \equiv p(v_1, v_2, \ldots, v_l)$, where $l \leq k$, $p \in \mathcal{P} - \{sm\}$. We have:

$$\begin{aligned}
&[\![p(v_1, v_2, \ldots, v_l)]\!]_3^S(Z) \\
&\quad = \iota^S(p)(Z(v_1), Z(v_2), \ldots, Z(v_l)) && \text{(Definition 3.4)} \\
&\quad = \bigsqcup_{f \circ Z^\natural = Z} \iota^{S^\natural}(p)(Z^\natural(v_1), Z^\natural(v_2), \ldots, Z^\natural(v_l)) && \text{(Definition 3.9)} \\
&\quad = \bigsqcup_{f \circ Z^\natural = Z} [\![p(v_1, v_2, \ldots, v_l)]\!]_3^{S^\natural}(Z^\natural) && \text{(Definition 3.4)} \\
&\quad = 1 && \text{(By equation (66))}
\end{aligned}$$

Notice that we use the fact that $p \not\equiv sm$ because the step from the second line to the third line may not hold for sm (cf. Definition 3.9).

Case 2: $b = 0$ and $a \equiv p(v_1, v_2, \ldots, v_l)$, where $l \leq k$, $p \in \mathcal{P} - \{sm\}$. We have:

$$\begin{aligned}
&[\![\neg p(v_1, v_2, \ldots, v_l)]\!]_3^S(Z) \\
&= 1 - \iota^S(p)(Z(v_1), Z(v_2), \ldots, Z(v_l)) && \text{(Definition 3.4)} \\
&= 1 - \bigsqcup_{f \circ Z^\natural = Z} \iota^{S^\natural}(p)(Z^\natural(v_1), Z^\natural(v_2), \ldots, Z^\natural(v_l)) && \text{(Definition 3.9)} \\
&= 1 - \bigsqcup_{f \circ Z^\natural = Z} [\![p(v_1, v_2, \ldots, v_l)]\!]_3^{S^\natural}(Z^\natural) && \text{(Definition 3.4)} \\
&= \bigsqcup_{f \circ Z^\natural = Z} [\![\neg p(v_1, v_2, \ldots, v_l)]\!]_3^{S^\natural}(Z^\natural) && \text{(Definition 3.4)} \\
&= 1 && \text{(By equation (66))}
\end{aligned}$$

Case 3: $b = 1$ and $a \equiv v_1 = v_2$, for $v_1 \not\equiv v_2$. We need to show that $Z(v_1) = Z(v_2)$ and $\iota(sm)(Z(v_1)) = 0$. If $Z(v_1) \neq Z(v_2)$ then there exists an assignment $Z^\natural$ such that $f \circ Z^\natural = Z$, and $Z^\natural(v_1) \neq Z^\natural(v_2)$ contradicting (66). Now assume that $\iota(sm)(Z(v_1)) = 1/2$ and thus by definition there exist $u_1, u_2 \in U^{S^\natural}$ such that $u_1 \neq u_2$ and $f(u_1) = f(u_2) = Z(v_1)$. Therefore, for $Z^\natural(v_1) = u_1$ and $Z^\natural(v_2) = u_2$, we get a contradiction to (66).

Case 4: $b = 0$ and $a \equiv v_1 = v_2$. We need to show that $Z(v_1) \neq Z(v_2)$. If $Z(v_1) = Z(v_2)$ then there exists an assignment $Z^\natural$ such that $f \circ Z^\natural = Z$, and $Z^\natural(v_1) = Z^\natural(v_2)$ contradicting (66).

Lemma B.1. *For every pair of structures $S_1, S_2 \in$ 3-CSTRUCT$[\mathcal{P}, \widehat{r}(F)]$ such that $U^{S_1} = U^{S_2} = U$, the structure $S_1 \sqcup S_2$ is also in 3-CSTRUCT$[\mathcal{P}, \widehat{r}(F)]$.*
Proof: By contradiction. Assume that constraint $\varphi_1 \rhd \varphi_2$ in $\widehat{r}(F)$ is violated. By definition, this happens when for some Z, $[\![\varphi_1]\!]_3^{S_1 \sqcup S_2}(Z) = 1$ and $[\![\varphi_2]\!]_3^{S_1 \sqcup S_2}(Z) \neq 1$. Because Kleene's semantics is monotonic in the information order (Lemma 3.7), $[\![\varphi_1]\!]_3^{S_1}(Z) = 1$ and $[\![\varphi_1]\!]_3^{S_2}(Z) = 1$. Therefore, because S_1 and S_2 both satisfy the constraint $\varphi_1 \rhd \varphi_2$, we have $[\![\varphi_2]\!]_3^{S_1}(Z) = 1$ and $[\![\varphi_2]\!]_3^{S_2}(Z) = 1$. But because φ_2 is an atomic formula or the negation of an atomic formula, $[\![\varphi_2]\!]_3^{S_1 \sqcup S_2}(Z) = 1$, which is a contradiction.

B.2 Correctness of the Coerce Algorithm

The correctness of algorithm Coerce stems from the following lemma:

Lemma B.2. *For every $S \in$ 3-STRUCT$[\mathcal{P}]$ and structure S' before each iteration of* Coerce(S)*, the following conditions hold: (i) $S' \sqsubseteq S$; (ii) If* Coerce *returns $\perp$ on S', then coerce$(S) = \perp$; and (iii) If coerce$(S) \neq \perp$, then coerce$(S) \sqsubseteq S$.*
Proof: By induction on the number of iterations.
Basis: For zero number of iterations the claim holds since (i) $S' = S$ and thus $S' \sqsubseteq S$ (i); (ii) Condition (ii) vacuously holds; (iii) If *coerce*$(S) \neq \perp$ then *coerce*$(S) \sqsubseteq S = S'$ and thus condition (iii) holds.
Induction hypothesis: Assume that the Lemma holds for $i \geq 0$ iterations.
Induction step: Let S' be the structure before the i-th iteration of Coerce and let us show that the lemma still holds after the i-th iteration.

Part I It is easy to see that in cases that Coerce does not return $\perp$, Coerce only lowers predicate values and therefore (i) holds after the i-iteration.

Part II Let us show that (ii) holds in the cases that Coerce returns $\perp$. Let us assume that $S'' =$ *coerce*$(S) \neq \perp$ and derive a contradiction. By the induction hypothesis, (i) $S'' \sqsubseteq S'$. Therefore by Lemma 3.7, $[\![\varphi_1]\!]_3^{S''}(Z) = 1$, and hence since $S'' \models \widehat{r}(F)$, it must be that $[\![\varphi_2]\!]_3^{S''}(Z) = 1$. The proof splits into the following cases when Coerce returns $\bot$:

Case 1: Coerce returns $\bot$ when Type I constraints in violated. Immediate.

Case 2: Coerce returns $\bot$ when Type II constraints in violated. There are two subcases to consider.

Case 2.1: $\varphi_2 \equiv v_1 = v_2$. Since $[\![v_1 = v_2]\!]_3^{S'}(Z) \neq 1$ and $\iota^{S'}(sm)(Z(v_1))$, by Definition 3.4, $Z(v_1) \neq Z(v_2)$, and therefore by Definition 3.4, $[\![v_1 = v_2]\!]_3^{S'}(Z) \neq 1$ — a contradiction.

Case 2.2: $\varphi_2 \equiv \neg(v_1 = v_2)$. Since $[\![\neg(v_1 = v_2)]\!]_3^{S'}(Z) \neq 1$, by Definition 3.4, $Z(v_1) = Z(v_2)$, and therefore by Definition 3.4, $[\![\neg(v_1 = v_2)]\!]_3^{S'}(Z) \neq 1$ — a contradiction.

Case 3: Coerce returns $\bot$ when Type III constraints in violated. Since $[\![p(v_1, v_2, \ldots, v_k)]\!]_3^{S'}(Z) \neq 1$, and $\iota^{S'}(p)(Z(v_1), Z(v_2), \ldots, Z(v_k)) \neq 1/2$. By Definition 3.4, $\iota^{S'}(p)(Z(v_1), Z(v_2), \ldots, Z(v_k)) = 1 - b$. By the induction hypothesis, (i) $S'' \sqsubseteq S'$. $\iota^{S''}(p)(Z(v_1), Z(v_2), \ldots, Z(v_k)) = 1 - b$ and therefore by Definition 3.4, $[\![p(v_1, v_2, \ldots, v_k)]\!]_3^{S'}(Z) \neq 1$, — a contradiction.

Part III Let us show that (iii) holds. Assume that $S'' = \mathit{coerce}(S) \neq \bot$. By the induction hypothesis, $S'' \sqsubseteq S'$. Therefore by Lemma 3.7, since $[\![\varphi_1]\!]_3^{S''}(Z) = 1$, and hence since $S'' \models \widehat{r}(F)$, $[\![\varphi_2]\!]_3^{S''}(Z) = 1$. The proof splits into the following cases:

Case 1: Coerce lowers $\iota^{S'}(sm)(Z(v_1))$ from 1/2 into 0 when $\varphi_2 \equiv v_1 = v_2$. Since $[\![v_1 = v_2]\!]_3^{S''}(Z) = 1$, by Definition 3.4, $Z(v_1) = Z(v_2)$ and $\iota^{S''}(sm)(Z(v_1)) = 1$ and therefore after the i-th iteration $S'' \sqsubseteq S'$.

Case 2: Coerce lowers $\iota^{S'}(p)(Z(v_1), Z(v_2), \ldots, Z(v_k))$ from 1/2 into b when $\varphi_2 \equiv p^b(v_1, v_2, \ldots, v_k)$. Since $[\![v_1 = v_2]\!]_3^{S''}(Z) = 1$, by Definition 3.4, $\iota^{S''}(p)(Z(v_1), Z(v_2), \ldots, Z(v_k)) = 1$, and therefore after the i-th iteration $S'' \sqsubseteq S'$.

Theorem 5.23 *For every $S \in$ 3-STRUCT[$\mathcal{P}$], $\mathit{coerce}(S) =$* Coerce(S).

Proof: Let T be the return value of Coerce(S). We discriminate between the following cases according to the value of T:

—Suppose $T = \bot$. By Lemma B.2, (ii), $\mathit{coerce}(S) = \bot = T$.

—If $T \neq \bot$, then by Lemma B.2, (i) $T \sqsubseteq S$. By the definition of the Coerce algorithm, $T \models \widehat{r}(F)$, and therefore, by Definition 5.20, $\mathit{coerce}(S) \neq \bot$. Hence, by Lemma B.2 (iii), $\mathit{coerce}(S) \sqsubseteq T$. Consequently, because $\mathit{coerce}(S)$ is the maximal structure that models $\widehat{r}(F)$, it must be that $\mathit{coerce}(S) = T$.