Parametric Shape Analysis via 3-Valued Logic

MOOLY SAGIV
and
THOMAS REPS
and
REINHARD WILHELM

Shape analysis concerns the problem of determining "shape invariants" for programs that perform destructive updating on dynamically allocated storage. This paper presents a *parametric* framework for shape analysis that can be instantiated in different ways to create different shape-analysis algorithms that provide varying degrees of efficiency and precision.

A key innovation of the work is that the stores that can possibly arise during execution are represented (conservatively) using 3-valued logical structures. The framework is instantiated in different ways by varying the predicates used in the 3-valued logic. The class of programs to which a given instantiation of the framework can be applied is not limited *a priori* (i.e., as in some work on shape analysis, to programs that manipulate only lists, trees, dags, etc.); each instantiation of the framework can be applied to any program, but may produce imprecise results (albeit conservative ones) due to the set of predicates employed.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures; dynamic storage management*; 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*

General Terms: Algorithms, Languages, Theory, Verification

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

---

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 carried out while Sagiv was at the Univ. of Chicago. Sagiv was supported in part by the National Science Foundation under grant CCR-9619219 and by the U.S.-Israel Binational Science Foundation under grant 96-00337. Address: Dept. of Comp. Sci.; School of Math. Sci.; Tel-Aviv Univ.; Tel-Aviv 69978; Israel. 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 U.S.-Israel Binational Science Foundation under grant 96-00337, by a Vilas Associate Award from the Univ. of Wisconsin, by the Office of Naval Research under contract N00014-00-1-0607, by the Alexander von Humboldt Foundation, and by the John Simon Guggenheim Memorial Foundation. Address: Comp. Sci. Dept.; Univ. of Wisconsin; 1210 W. Dayton St.; Madison, WI 53706; USA. E-mail: reps@cs.wisc.edu.

Wilhelm was supported in part by a DAAD-NSF Collaborative Research Grant. Address: Fachrichtung Inf.; Univ. des Saarlandes; 66123 Saarbrücken; Ger. E-mail: wilhelm@cs.uni-sb.de.

1. INTRODUCTION

In the past two decades, many "shape-analysis" algorithms have been developed that can automatically create different classes of "shape descriptors" for programs that perform destructive updating on dynamically 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 the set of possible memory states ("stores") that arise at a given point in the program by *shape graphs*, in which heap cells are represented by shape-graph nodes and, in particular, sets of "indistinguishable" heap cells are represented by a single shape-graph node (often called a *summary-node* [Chase et al. 1990]).

This paper presents a *parametric* framework for shape analysis. The framework can be instantiated in different ways to create shape-analysis algorithms that provide different degrees of precision. The essence of 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 this framework. Other instantiations of the framework yield new shape-analysis algorithms that obtain more precise information than previous work.

A parametric framework must address two issues:

i. What is the language for specifying (a) the properties of stores that are to be tracked, and (b) how such properties are affected by the execution of the different kinds of statements in the programming language?

ii. How is a shape-analysis algorithm generated from such a specification?

Issue *i.* concerns the *specification language* of the framework. A key innovation of our work is the way in which it makes use of 2-valued and 3-valued logic: 2-valued and 3-valued logical structures are used to represent concrete and abstract stores, respectively (i.e., interpretations of unary and binary predicates encode the contents of variables and pointer-valued structure fields); first-order formulae with transitive closure are used to specify properties such as sharing, cyclicity, reachability, etc. Formulae are also used to specify how the store is affected by the execution of the different kinds of statements in the programming language. The analysis framework can be instantiated in different ways by varying the predicates that are used. The specified set of predicates determines the set of data-structure properties that can be tracked, and consequently what properties of stores can be "discovered" to hold at the different points in the program by the corresponding instance of the analysis. Issue *ii.* concerns how to create an actual analyzer from the specification. In our work, the analysis algorithm is an abstract interpretation; it finds the least fixed point of a set of equations that are generated from the analysis specification.

The ideal is to have a fully automatic parametric framework—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 been implemented by T. Lev-Ami [Lev-Ami 2000; Lev-Ami and Sagiv 2000]. (See also Sect. 7.4.1.)

The class of programs to which a given instantiation of the framework can be applied is not limited *a priori* (i.e., as in some work on shape analysis, to programs that manipulate only lists, trees, dags, etc.). Each instantiation of the framework can be applied to any program, but may produce conservative results due to the set of predicates employed; that is, the attempt to analyze a particular program with an instantiation created using an inappropriate set of predicates may produce imprecise, albeit conservative, results. Thus, depending on the kinds of linked data structures used in a program, and on the link-rearrangement operations performed by the program's statements, a different instantiation—using a different set of predicates—may be needed in order to obtain more useful results.

The framework allows one to create algorithms that are more precise than the shape-analysis algorithms cited earlier. 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 Sects. 2.6 and 5.3). Other static-analysis techniques yield very imprecise information on these programs. So that reachability properties can be specified, the specification language of the framework includes a transitive-closure operator.

The key features of the approach described in this paper are as follows:

—The use of 2-valued logical structures to represent concrete stores. Interpretations of unary and binary predicates encode the contents of variables and pointer-valued structure fields (see Sect. 2.2).

—The use of a 2-valued first-order logic with transitive closure to specify properties of stores such as sharing, cyclicity, reachability, etc. (see Sects. 2, 3, and 5).

—The use of Kleene's 3-valued logic [Kleene 1987] to relate the concrete (2-valued) world and the abstract (3-valued) world. Kleene's logic has a third truth value that signifies "unknown", which is useful for shape analysis because we only have partial information about summary nodes. For these nodes, predicates may have the value *unknown*. (See Sects. 2 and 4.)

—The development of a systematic way to construct abstract domains that are suitable for shape analysis. This is based on a general notion of "truth-blurring" embeddings that map from a 2-valued world to a corresponding 3-valued one (see Sects. 2.5, 4.2, and 4.3).

—The use of the Embedding Theorem (Theorem 4.9) to ensure 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. The consequence of the Embedding Theorem is that it allows us to extract information from *either* the concrete world or the abstract world via a *single* formula—the same syntactic expression can be interpreted either in the 2-valued world or the 3-valued world. The consistency of the information obtained is ensured by the Embedding Theorem. This eases soundness proofs considerably.

—New insight into the issue of "materialization". This is known to be very important for maintaining accuracy in the analysis of loops that advance pointers through linked data structures [Chase et al. 1990; Plevyak et al. 1993; Sagiv et al. 1998]. (Materialization involves the splitting of a summary-node into two separate nodes by the abstract transfer function that expresses the semantics of a statement of the form `x = y->n`.) This paper develops a new approach to materialization:

—The essence of materialization involves a step (called *focus* in Sect. 6.3) that forces the values of certain formulae from *unknown* to *true* or *false*. This has the effect of converting an abstract store into several abstract stores, each of which is more precise than the original one.

—Materialization is complicated because various properties of a store are interdependent. We introduce a mechanism based on a constraint-satisfaction system to capture the effects of such dependences (see Sect. 6.4).

In this paper, we address the problem of shape analysis for a single procedure. This has allowed us to concentrate on foundational aspects of shape-analysis methods. The application of our techniques to the problem of interprocedural shape analysis, including shape analysis for programs with recursive procedures, is addressed in [Rinetskey and Sagiv 2001] (see also Sect. 7.4.3).

The remainder of the paper is organized as follows: Sect. 2 provides an overview of the shape-analysis framework. Sect. 3 shows how 2-valued logic can be used as a meta-language for expressing the concrete operational semantics of programs (and programming languages). Sect. 4 provides the technical details about the representation of stores using 3-valued logic. Sect. 5 defines the notion of instrumentation predicates, which are used to specify the abstract domain that a specific instantiation of the shape-analysis framework will use. Sect. 6 formulates the abstract semantics for program statements and conditions, and defines the iterative algorithm for computing a (safe) set of 3-valued structures for each program point. Sect. 7 discusses related work. Sect. 8 makes some final observations.

In Appendix A, we sketch how the framework can be used to analyze programs that manipulate doubly linked lists, which has been posed as a challenging problem for program analysis [Aiken 1996]. The proof of the Embedding Theorem and other technical proofs are presented in Appendices B and C.

2. AN OVERVIEW OF THE PARAMETRIC FRAMEWORK

This section provides an overview of the main ideas used in the paper. The presentation is at a semi-technical level; a more detailed treatment of this material, as well as several elaborations on the ideas covered here, is presented in the later sections of the paper.

2.1 Shape Invariants and Data Structures

Constituents of shape invariants that can be used to characterize a data structure include

i. anchor pointer variables, i.e., information about which pointer variables point into the data structure;

ii. the types of the data-structure elements, and in particular, which fields hold pointers;

iii. connectivity properties, such as

—whether or not all elements of the data structure are reachable from a root pointer variable,

—whether or not any data-structure elements are shared,
—whether or not there are cycles in the data structure,
—whether or not an element v pointed to by a "forward" pointer of another element v' has its "backward" pointer pointing to v';

iv. other properties, for instance, whether an element of an ordered list is in the correct position.

Each data structure can be characterized by a certain set of such properties.

Most semantics track the values of pointer variables and pointer-valued fields using a pair of functions, often called the *environment* and the *store*. Constituents *i.* and *ii.* above are parts of any such semantics; consequently, we refer to them as *core* properties.

Connectivity and other properties, such as those mentioned in *iii.* and *iv.*, are usually not explicitly part of the semantics of pointers in a language, but instead are properties derived from this core semantics. They are essential ingredients in program verification, however, as well as in our approach to shape analysis of programs. Non-core properties will be called *instrumentation* properties (for reasons that will become clear shortly).

Let us start by taking a Platonic view, namely that ideas exist without regard to their physical realization. Concepts like "is shared", "lies on a cycle", and "is reachable" can be defined either in graph-theoretic terms, using properties of paths, or in terms of the programming-language concept of pointers. The definitions of these concepts can be stated in a way that is independent of any particular data structure; for instance:

EXAMPLE 2.1. A heap cell is *heap-shared* if it is the target of two pointers—either from two different heap cells, or from two different pointer components of the same heap cell.

Data structures can now be characterized using sets of such properties, where "data structure" here is still independent of a particular implementation; for instance:

EXAMPLE 2.2. An *acyclic singly linked list* is a set of objects, each with one pointer component. The objects are *reachable from a root pointer variable* either directly or by following pointer components. No object *lies on a cycle*, i.e., is reachable from itself by following pointer components.

To address the problem of verifying or analyzing a particular program that uses a certain data structure, we have to leave the Platonic realm, and formulate shape invariants in terms of the pointer variables and data-type declarations from that program.

EXAMPLE 2.3. Fig. 1(a) shows the declaration of a linked-list data type in C, and Fig. 1(b) shows a C program that searches a list and splices a new element into the list. The characterization of an acyclic singly linked list in terms of the properties "is reachable from a root pointer variable" and "lies on a cycle" can now be specialized for that data-type declaration and that program as follows:

—"is reachable from a root pointer variable" means "is reachable from `x`, or is reachable from `y`, or is reachable from `t`, or is reachable from `e`".

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

(a)

```
/* insert.c */
#include ‘‘list.h’’
void insert(List x, int d) {
  List y, t, e;
  assert(acyclic_list(x) && x != NULL);
  y = x;
  while (y->n != NULL && ...) {
    y = y->n;
  }
  t = malloc();
  t->d = d;
  e = y->n;
  t->n = e;
  y->n = t;
}
```

(b)

Fig. 1. (a) Declaration of a linked-list data type in C. (b) A C function that searches a list pointed to by parameter `x`, and splices in a new element.

—“lies on a cycle” means “is reachable from itself following one or more `n` fields”.

To be able to carry out shape analysis, a number of additional concepts need to be formalized:

—An encoding (or representation) of stores, so that we can talk precisely about store elements and the relationships among them.

—A language in which to state properties that store elements may or may not possess.

—A way to extract the properties of stores and store elements.

—A definition of the concrete semantics of the programming language—and, in particular, one that makes it possible to track how properties change as the execution of a program statement changes the store.

—A technique for creating abstractions of stores so that abstract interpretation can be applied.

In our approach, the formalization of each of these concepts is based on predicate logic.

2.2 Representing Stores via 2-Valued and 3-Valued Logical Structures

To represent stores, we work with what logicians call *logical structures*. A logical structure is associated with a *vocabulary* of predicate symbols (with given arities); each logical structure S, denoted by $\langle U^S, \iota^S \rangle$, has a universe of *individuals* U^S. In a 2-valued logical structure, ι^S maps each arity-k predicate symbol p and possible k-tuple of individuals $(u_1, \ldots, u_k)$, where $u_i \in U^S$, to the value 0 or 1 (i.e., *false* and *true*, respectively). In a 3-valued logical structure, ι^S maps p and $(u_1, \ldots, u_k)$ to the value 0, 1, or 1/2 (i.e., *false*, *true*, and *unknown*, respectively).

2-valued logical structures will be used to encode concrete stores; 3-valued logical structures will be used to encode abstract stores; members of these two families of

Predicate	**Intended Meaning**
$q(v)$	Does pointer variable `q` point to element v?
$n(v_1, v_2)$	Does the `n` field of v_1 point to v_2?

Table I. Predicates used for representing the stores manipulated by programs that use the `List` data-type declaration from Fig. 1(a).

structures will be related by "truth-blurring embeddings" (which are explained in Sect. 2.5).

2-valued logical structures are used to encode concrete stores as follows: Individuals represent memory locations in the heap; pointers from the stack into the heap are represented by unary "pointed-to-by-variable-`q`" predicates; and pointer-valued fields of data structures are represented by binary predicates.

EXAMPLE 2.4. Table I lists the predicates used for representing the stores manipulated by programs that use the `List` data-type declaration from Fig. 1(a). In the case of `insert`, the unary predicates x, y, t, and e correspond to the program variables `x`, `y`, `t`, and `e`, respectively. The binary predicate n corresponds to the `n` fields of `List` elements.

Fig. 2 illustrates the 2-valued logical structures that represent lists of length ≤ 4 that are pointed to by program variable `x`. In column 3 of Fig. 2, the following graphical notation is used for depicting 2-valued logical structures:

- Individuals of the universe are represented by circles with names inside.
- A unary predicate p is represented in the graph by having a solid arrow from the predicate name p to node u for each individual u for which $\iota(p)(u) = 1$, and no arrow from predicate name p to node u' for each individual u' for which $\iota(p)(u') = 0$. (If $\iota(p)$ is 0 for all individuals, the predicate name p will not be shown.)
- A binary predicate q is represented in the graph by having a solid arrow labeled q between each pair of individuals u_i and u_j for which $\iota(q)(u_i, u_j) = 1$, and no arrow between pairs u'_i and u'_j for which $\iota(q)(u'_i, u'_j) = 0$.

Thus, in structure $S_2^\natural$, pointer variable `x` points to individual u_1, whose `n` field points to individual u_2.[1] The `n` field of u_2 does not point to any individual (i.e., u_2 represents a heap cell whose `n` field has the value `NULL`).

Throughout Sect. 2, all examples of structures show both the tables of unary and binary predicates, as well as the corresponding graphical representation. In all other sections of the paper, the tables are omitted and just the graphical representation is shown.

2.3 Extraction of Store Properties

2-valued structures offer a systematic way to answer questions about properties of the concrete stores they encode. As an example, consider the formula

$$\varphi_{is}(v) \stackrel{\text{def}}{=} \exists v_1, v_2 : n(v_1, v) \land n(v_2, v) \land v_1 \neq v_2, \tag{1}$$

---

[1] 2-valued logical structures are indicated by superscripting them with the "natural" symbol ($\natural$).

<table>
<tr><th>Name</th><th>Logical Structure</th><th>Graphical Representation</th></tr>
<tr><td>$S_0^\natural$</td><td>

unary preds.

<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr></table>

binary preds.

<table><tr><th>n</th></tr></table>

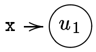
</td><td></td></tr>
<tr><td>$S_1^\natural$</td><td>

unary preds.

<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr><tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>

binary preds.

<table><tr><th>n</th><th>u_1</th></tr><tr><td>u_1</td><td>0</td></tr></table>

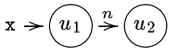
</td><td></td></tr>
<tr><td>$S_2^\natural$</td><td>

unary preds.

<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr><tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>

binary preds.

<table><tr><th>n</th><th>u_1</th><th>u_2</th></tr><tr><td>u_1</td><td>0</td><td>1</td></tr><tr><td>u_2</td><td>0</td><td>0</td></tr></table>

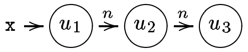
</td><td></td></tr>
<tr><td>$S_3^\natural$</td><td>

unary preds.

<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr><tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>

binary preds.

<table><tr><th>n</th><th>u_1</th><th>u_2</th><th>u_3</th></tr><tr><td>u_1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>1</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td></tr></table>

</td><td></td></tr>
<tr><td>$S_4^\natural$</td><td>

unary preds.

<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr><tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>

binary preds.

<table><tr><th>n</th><th>u_1</th><th>u_2</th><th>u_3</th><th>u_4</th></tr><tr><td>u_1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>u_4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>

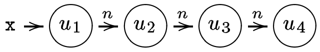
</td><td></td></tr>
</table>

Fig. 2. The 2-valued logical structures that represent lists of length ≤ 4.

which expresses the "is-shared" property: "Do two or more different heap cells point to heap cell v via their n fields?" For instance, $\varphi_{is}(v)$ evaluates to 0 in $S_2^\natural$ for $v \mapsto u_2$, because there is no assignment $v_1 \mapsto u_i$ and $v_2 \mapsto u_j$ such that $\iota^{S_2^\natural}(n)(u_i, u_2)$, $\iota^{S_2^\natural}(n)(u_j, u_2)$, and $u_i \neq u_j$ all hold. As a second example, consider the formula

$$\varphi_{c_n}(v) \stackrel{\text{def}}{=} n^+(v, v) \tag{2}$$

which expresses the property of whether a heap cell v appears on a directed n-cycle. Here n^+ denotes the transitive closure of the n-relation. Formula $\varphi_{c_n}(v)$ evaluates to 0 in $S_2^\natural$ for $v \mapsto u_2$, because the transitive closure of the relation $\iota^{S_2^\natural}(n)$ does not contain the pair (u_2, u_2).

The preceding discussion can be summarized as the following principle:

OBSERVATION 2.5. [**Property-Extraction Principle**]. *By encoding stores as logical structures, questions about properties of stores can be answered by evaluating formulae. The property holds or does not hold, depending on whether the formula evaluates to* 1 *or* 0*, respectively, in the logical structure.*

2.4 Expressing the Semantics of Program Statements

Our tool for expressing the semantics of program statements is also based on evaluating formulae:

OBSERVATION 2.6. [**Expressing the Semantics of Statements via Logical Formulae**]. *Suppose that σ is a store that arises before statement st, that σ' is the store that arises after st is evaluated on σ, and that S is the logical structure that*

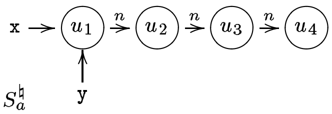
Structure Before

unary preds.

indiv.	x	y	t	e
u_1	1	1	0	0
u_2	0	0	0	0
u_3	0	0	0	0
u_4	0	0	0	0

binary preds.

n	u_1	u_2	u_3	u_4
u_1	0	1	0	0
u_2	0	0	1	0
u_3	0	0	0	1
u_4	0	0	0	0



$S^{\natural}_a$

Statement: `y = y->n`

Predicate Update Formulae

$$x'(v) = x(v)$$
$$y'(v) = \exists v_1 : y(v_1) \wedge n(v_1, v)$$
$$t'(v) = t(v)$$
$$e'(v) = e(v)$$
$$n'(v_1, v_2) = n(v_1, v_2)$$

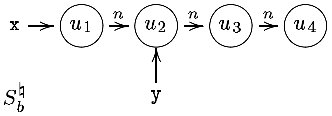
Structure After

unary preds.

indiv.	x	y	t	e
u_1	1	0	0	0
u_2	0	1	0	0
u_3	0	0	0	0
u_4	0	0	0	0

binary preds.

n	u_1	u_2	u_3	u_4
u_1	0	1	0	0
u_2	0	0	1	0
u_3	0	0	0	1
u_4	0	0	0	0



$S^{\natural}_b$

Fig. 3. The given predicate-update formulae express a transformation on logical structures that corresponds to the semantics of `y = y->n`.

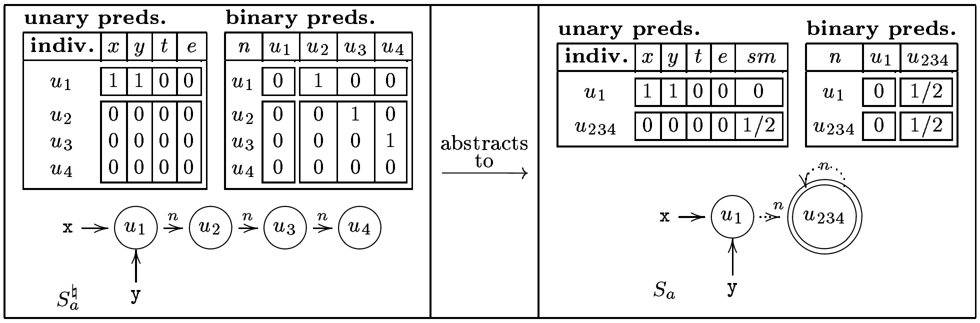
encodes σ. A collection of **predicate-update formulae***—one for each predicate p in the vocabulary of S—allows one to obtain the structure S' that encodes σ': When evaluated in structure S, the predicate-update formula for a predicate p indicates what the value of p should be in S'.*

In other words, the set of predicate-update formulae captures the concrete semantics of *st*.

This process is illustrated in Fig. 3 for the statement `y = y->n`, where the initial structure $S^{\natural}_a$ represents a list of length 4 that is pointed to by both `x` and `y`. Fig. 3 shows the predicate-update formulae for the five predicates of the vocabulary used in conjunction with `insert`: x, y, t, e, and n; the symbols x', y', t', e', and n' denote the values of the corresponding predicates in the structure that arises after execution of `y = y->n`. Predicates x', t', e', and n' are unchanged in value by `y = y->n`. The predicate-update formula $y'(v) = \exists v_1 : y(v_1) \wedge n(v_1, v)$ expresses the advancement of program variable `y` down the list.

2.5 Abstraction Via Truth-Blurring Embeddings

The abstract stores used for shape-analysis are 3-valued logical structures that, by the construction discussed below, are *a priori* of bounded size. In general, each 3-valued logical structure corresponds to a (possibly infinite) set of 2-valued logical



unary preds.

indiv.	x	y	t	e
u_1	1	1	0	0
u_2	0	0	0	0
u_3	0	0	0	0
u_4	0	0	0	0

binary preds.

n	u_1	u_2	u_3	u_4
u_1	0	1	0	0
u_2	0	0	1	0
u_3	0	0	0	1
u_4	0	0	0	0

abstracts to

unary preds.

indiv.	x	y	t	e	sm
u_1	1	1	0	0	0
u_{234}	0	0	0	0	1/2

binary preds.

n	u_1	u_{234}
u_1	0	1/2
u_{234}	0	1/2

Fig. 4. The abstraction of 2-valued structure $S^{\natural}_a$ to 3-valued structure S_a. The boxes in the tables of unary predicates indicate how individuals are grouped into equivalence classes; the boxes in the tables for predicate n indicate how the quotient of n with respect to these equivalence classes is performed.

structures. Members of these two families of structures are related by "truth-blurring embeddings".

The principle behind truth-blurring embedding is illustrated in Fig. 4, which shows how 2-valued structure $S^{\natural}_a$ is abstracted to 3-valued structure S_a. Abstraction is driven by the values of the "vector" of unary predicate values that each individual u has—i.e., for $S^{\natural}_a$, by the values $\iota(x)(u)$, $\iota(y)(u)$, $\iota(t)(u)$ and $\iota(e)(u)$—and, in particular, by the equivalence classes formed from the individuals that have the same vector for their unary predicate values. In $S^{\natural}_a$, there are two such equivalence classes: (i) $\{u_1\}$, for which x, y, t, and e are 1, 1, 0, and 0, respectively, and (ii) $\{u_2, u_3, u_4\}$, for which x, y, t, and e are all 0. (The boxes in the table of unary predicates for $S^{\natural}_a$ show how individuals of $S^{\natural}_a$ are grouped into two equivalence classes.)

All members of such equivalence classes are mapped to the same individual of the 3-valued structure. Thus, all members of $\{u_2, u_3, u_4\}$ from $S^{\natural}_a$ are mapped to the same individual in S_a, called u_{234};[2] similarly, all members of $\{u_1\}$ from $S^{\natural}_a$ are mapped to the same individual in S_a, called u_1.

For each non-unary predicate of the 2-valued structure, the corresponding predicate in the 3-valued structure is formed by a "truth-blurring quotient". For instance,

—In $S^{\natural}_a$, $\iota^{S^{\natural}_a}(n)$ evaluates to 0 for the only pair of individuals in $\{u_1\} \times \{u_1\}$. Therefore, in S_a the value of $\iota^{S_a}(n)(u_1, u_1)$ is 0.

—In $S^{\natural}_a$, $\iota^{S^{\natural}_a}(n)$ evaluates to 0 for all pairs from $\{u_2, u_3, u_4\} \times \{u_1\}$. Therefore, in S_a the value of $\iota^{S_a}(n)(u_{234}, u_1)$ is 0.

—In $S^{\natural}_a$, $\iota^{S^{\natural}_a}(n)$ evaluates to 0 for two of the pairs from $\{u_1\} \times \{u_2, u_3, u_4\}$ (i.e., $\iota^{S^{\natural}_a}(n)(u_1, u_3) = 0$ and $\iota^{S^{\natural}_a}(n)(u_1, u_4) = 0$), whereas $\iota^{S^{\natural}_a}(n)$ evaluates to 1 for the

---

[2] The reader should bear in mind that the names of individuals are completely arbitrary: u_{234} could have been called u_{17} or u_{99}, etc.; in particular, the subscript "234" is used here only to remind the reader that, in this example, u_{234} of S_a is the individual that represents $\{u_2, u_3, u_4\}$ of $S^{\natural}_a$. (In many subsequent examples, u_{234} will be named u.)

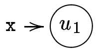
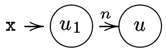
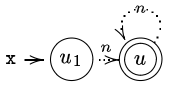
<table>
<tr><th>Name</th><th>Logical Structure</th><th>Graphical Representation</th></tr>
<tr><td>S_0</td><td>unary preds.<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th><th>sm</th></tr></table>binary preds.<table><tr><th>n</th></tr></table></td><td></td></tr>
<tr><td>S_1</td><td>unary preds.<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th><th>sm</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 preds.<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 preds.<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th><th>sm</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>0</td><td>0</td></tr></table>binary preds.<table><tr><th>n</th><th>u_1</th><th>u</th></tr><tr><td>u_1</td><td>0</td><td>1</td></tr><tr><td>u</td><td>0</td><td>0</td></tr></table></td><td> </td></tr>
<tr><td>S_3</td><td>unary preds.<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th><th>sm</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>0</td><td>1/2</td></tr></table>binary preds.<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. 5. The 3-valued logical structures that are obtained by applying truth-blurring embedding to the 2-valued structures that appear in Fig. 2.

other pair (i.e., $\iota^{S^\natural_a}(n)(u_1, u_2) = 1$); therefore, in S_a the value of $\iota^{S_a}(n)(u_1, u_{234})$ is 1/2.

—In $S^\natural_a$, $\iota^{S^\natural_a}(n)$ evaluates to 0 for some pairs from $\{u_2, u_3, u_4\} \times \{u_2, u_3, u_4\}$ (e.g., $\iota^{S^\natural_a}(n)(u_2, u_4) = 0$), whereas $\iota^{S^\natural_a}(n)$ evaluates to 1 for other pairs (e.g., $\iota^{S^\natural_a}(n)(u_2, u_3) = 1$); therefore, in S_a the value of $\iota^{S_a}(n)(u_{234}, u_{234})$ is 1/2.

In Fig. 4, the boxes in the tables for predicate `n` indicate these four groupings of values.

An additional unary predicate, called *sm* (standing for "summary"), is added to the 3-valued structure to capture whether individuals of the 3-valued structure represent more than one concrete individual. For instance, $\iota^{S_a}(sm)(u_1) = 0$ because u_1 in S_a represents a single individual of $S^\natural_a$. On the other hand, u_{234} represents three individuals of $S^\natural_a$. For technical reasons, *sm* can be 0 or 1/2, but never 1; therefore, $\iota^{S_a}(sm)(u_{234}) = 1/2$.

The graphical notation for 3-valued logical structures (cf. structure S_a of Fig. 4) is derived from the one for 2-valued structures, with the following additions:

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

—Unary and binary predicates with value 1/2 are represented by dotted arrows.

Thus, in structure S_a of Fig. 4, pointer variables `x` and `y` definitely point to the concrete element represented by u_1, whose `n` field may point to a concrete element represented by element u_{234}; u_{234} is a summary node, i.e., it may represent more than one concrete element. Possibly there is an `n` field in one or more of these concrete elements that points to another of the concrete elements represented by u_{234}, but there cannot be an `n` field in any of these concrete elements that points to the concrete element represented by u_1.

Fig. 5 shows the 3-valued structures that are obtained by applying truth-blurring embedding to the 2-valued structures that appear in Fig. 2. In addition to the lists of length 3 and 4 from Fig. 2 (i.e., $S^\natural_3$ and $S^\natural_4$), the 3-valued structure S_3 also represents

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

- the acyclic lists of length 5, 6, etc. that are pointed to by `x`;
- the cyclic lists of length 3 or more that are pointed to by `x`, such that the backpointer is not to the head of the list, but to the second, third, or later element.

Thus, S_3 is a finite abstract structure that captures an infinite set of (possibly cyclic) concrete lists.

The structures S_0, S_1, and S_2 represent the cases of acyclic lists of length zero, one, and two, respectively.

2.6 Conservative Extraction of Store Properties

Kleene's 3-valued interpretation of the propositional operators is given in Table II. In Sect. 4.2, we give the *Embedding Theorem* (Theorem 4.9), 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. Thus, 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 does not hold in any store represented by the 3-valued structure.
- If a formula evaluates to 1/2, then we do not know if this formula holds in all stores, does not hold in any store, or holds in some stores and does not hold in some other stores represented by the 3-valued structure.

Consider the formula $\varphi_{c_n}(v)$ defined in Eqn. (2). ("Does heap cell v appear on a directed cycle of `n` fields?") Formula $\varphi_{c_n}(v)$ evaluates to 0 in S_3 for $v \mapsto u_1$, because $n^+(u_1, u_1)$ evaluates to 0 in Kleene's semantics.

Formula $\varphi_{c_n}(v)$ evaluates to 1/2 in S_3 for $v \mapsto u$: $n^+(u, u)$ evaluates to 1/2 because (i) $\iota^{S_3}(n)(u, u) = 1/2$ and (ii) there is no path of length one or more from u to u in which all edges have the value 1. Because of this, the evaluation of the formula does not tell us whether the elements that u represents do or do not lie on a cycle—some may and some may not. This uncertainty implies that (the tail of) the list pointed to by `x` *might* be cyclic.

In many situations, however, we are interested in analyzing the behavior of a program under the assumption, for example, that the program's input is an acyclic list. If an abstraction is not capable of expressing the distinction between cyclic and acyclic lists, an analysis algorithm based on that abstraction will usually be able to recover only very imprecise information about the actions of the program.

For this reason, we are interested in having our parametric framework support abstractions in which, for instance, the acyclic lists are distinguished from the cyclic

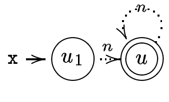
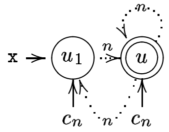
<table>
<tr><th>Name</th><th colspan="2">Logical Structure</th><th>Graphical Representation</th></tr>
<tr><td>$S_{acyclic}$</td><td>unary preds.
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th><th>sm</th><th>c_n</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1/2</td><td>0</td></tr>
</table></td><td>binary preds.
<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>S_{cyclic}</td><td>unary preds.
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th><th>sm</th><th>c_n</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>u</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1/2</td><td>1</td></tr>
</table></td><td>binary preds.
<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. 6. The 3-valued logical structures that are obtained by applying truth-blurring embedding—using x, y, t, e, and the cyclicity instrumentation predicate c_n—to the 2-valued structures that represent acyclic and cyclic lists of length 3 or more.

lists. Our framework supports such distinctions by using unary *instrumentation predicates*.

Example 2.7. Fig. 6 shows the 3-valued logical structures that are obtained by applying truth-blurring embedding—using x, y, t, e, and the cyclicity instrumentation predicate c_n—to the 2-valued structures that represent acyclic and cyclic lists of length 3 or more.

- In $S_{acyclic}$, we have $\iota^{S_{acyclic}}(c_n)(u_1) = 0$ and $\iota^{S_{acyclic}}(c_n)(u) = 0$. This implies that $S_{acyclic}$ can only represent acyclic lists *even though formula (2) evaluates to* 1/2 *on* u.
- In S_{cyclic}, the fact that the value of $\iota^{S_{cyclic}}(c_n)(u_1)$ is 1 indicates that u_1 definitely lies on a cycle, even though formula (2) evaluates to 1/2 on u_1. In addition, $\iota^{S_{cyclic}}(c_n)(u) = 1$, even though formula (2) evaluates to 1/2 on u, which indicates that all elements of the tails of the lists that S_{cyclic} represents lie on a cycle as well.

The preceding discussion illustrates the following principle:

Observation 2.8. [**Instrumentation Principle**]. *Suppose that* S *is a* 3*-valued structure that represents the* 2*-valued structure* $S^\natural$. *By explicitly "storing" in* S *the values that a formula* φ *has in* $S^\natural$, *it is sometimes possible to extract more precise information from* S *than can be obtained just by evaluating* φ *in* S.

In Sect. 5, several other instrumentation predicates are introduced that are useful both for analyzing data structures other than singly linked lists, as well as for increasing the precision of shape-analysis algorithms. By using the right collection of instrumentation predicates, 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.

As discussed further in Sect. 5.3, instrumentation predicates that track information about *reachability from pointer variables* are particularly important for avoiding a loss of precision, because they permit the abstract representations of data structures—and different parts of the same data structure—that are disjoint

<table>
<tr><th>Name</th><th>Logical Structure</th><th>Graphical Representation</th></tr>
<tr><td>$S_6^\natural$</td><td>

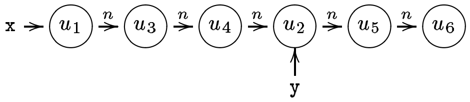
unary preds.

<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>u_2</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>u_3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>u_4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>u_5</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>u_6</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

binary preds.

<table>
<tr><th>n</th><th>u_1</th><th>u_2</th><th>u_3</th><th>u_4</th><th>u_5</th><th>u_6</th></tr>
<tr><td>u_1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>u_2</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>u_3</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>u_4</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>u_5</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>u_6</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

</td><td>



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

Fig. 7. A 2-valued structure for a list pointed to by `x`, where `y` points into the middle of the list.

in the concrete world to be kept separate [Sagiv et al. 1998, p. 38]. A reachability instrumentation predicate $r_{q,n}(v)$ captures whether v is (transitively) reachable from pointer variable `q` along `n` fields. This is illustrated in Figs. 7 and 8, which show how a concrete list in which `x` points to the head and `y` points into the middle is mapped to two different 3-valued structures, depending on whether the instrumentation predicates $r_{x,n}$, $r_{y,n}$, $r_{t,n}$, and $r_{e,n}$ are used or not. Note that the situation depicted in Fig. 7 is one that occurs in `insert` as `y` is advanced down the list. The reachability instrumentation predicates play a crucial role in developing a shape-analysis algorithm that is capable of obtaining precise shape information for `insert`.

2.7 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.

The fact that the semantics of statements can be expressed via logical formulae (Obs. 2.6), together with the fact that the evaluation of a formula φ in a 3-valued structure S is guaranteed to be safe with respect to the evaluation of φ in any 2-valued structure that S represents (the Embedding Theorem) means that one abstract semantics falls out automatically from the concrete semantics: one merely has to evaluate the predicate-update formulae of the concrete semantics on 3-valued structures.

OBSERVATION 2.9. [**Reinterpretation Principle**]. *Evaluation of the predicate-update formulae for a statement st in* 2*-valued logic captures the transfer function for st of the concrete semantics. Evaluation of the same formulae in* 3*-valued logic captures the transfer function for st of the abstract semantics.*

Fig. 9 combines Fig. 3 and Fig. 4 (see col. 2 and row 1 of Fig. 9, respectively).

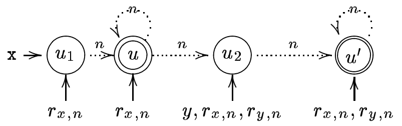
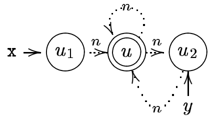
<table>
<tr><th>Name</th><th>Logical Structure</th><th>Graphical Representation</th></tr>
<tr><td>S_{reach}</td><td>unary preds.
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th><th>sm</th><th>c_n</th><th>$r_{x,n}$</th><th>$r_{y,n}$</th><th>$r_{t,n}$</th><th>$r_{e,n}$</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</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>0</td><td>1/2</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>u_2</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>u'</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1/2</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
</table>
binary preds.
<table>
<tr><th>n</th><th>u_1</th><th>u</th><th>u_2</th><th>u'</th></tr>
<tr><td>u_1</td><td>0</td><td>1/2</td><td>0</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td><td>1/2</td><td>0</td></tr>
<tr><td>u_2</td><td>0</td><td>0</td><td>0</td><td>1/2</td></tr>
<tr><td>u'</td><td>0</td><td>0</td><td>0</td><td>1/2</td></tr>
</table></td><td>
</td></tr>
<tr><td>S_{middle}</td><td>unary preds.
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th><th>sm</th><th>c_n</th></tr>
<tr><td>u_1</td><td>1</td><td>0</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>0</td><td>1/2</td><td>0</td></tr>
<tr><td>u_2</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>
binary preds.
<table>
<tr><th>n</th><th>u_1</th><th>u</th><th>u_2</th></tr>
<tr><td>u_1</td><td>0</td><td>1/2</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td><td>1/2</td></tr>
<tr><td>u_2</td><td>0</td><td>1/2</td><td>0</td></tr>
</table></td><td>
</td></tr>
</table>

Fig. 8. The 3-valued logical structures that are obtained by applying truth-blurring embedding—with and without reachability instrumentation predicates $r_{x,n}$, $r_{y,n}$, $r_{t,n}$, and $r_{e,n}$—to the 2-valued structure $S_6^\natural$ from Fig. 7.

Col. 4 of Fig. 9 illustrates how the predicate-update formulae that express the concrete semantics for `y = y->n` also express a transformation on 3-valued logical structures—i.e., an abstract semantics—that is safe with respect to the concrete semantics (cf. $S_a^\natural \to S_b^\natural$ versus $S_a \to S_b$).[3] To keep things simple, the issue of how to update the values of instrumentation predicates is not addressed here (see Sect. 5).

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

—Because the number of elements in the 3-valued structures that we work with is bounded, the abstract-interpretation process always terminates.

—The Embedding Theorem implies that the results obtained are conservative.

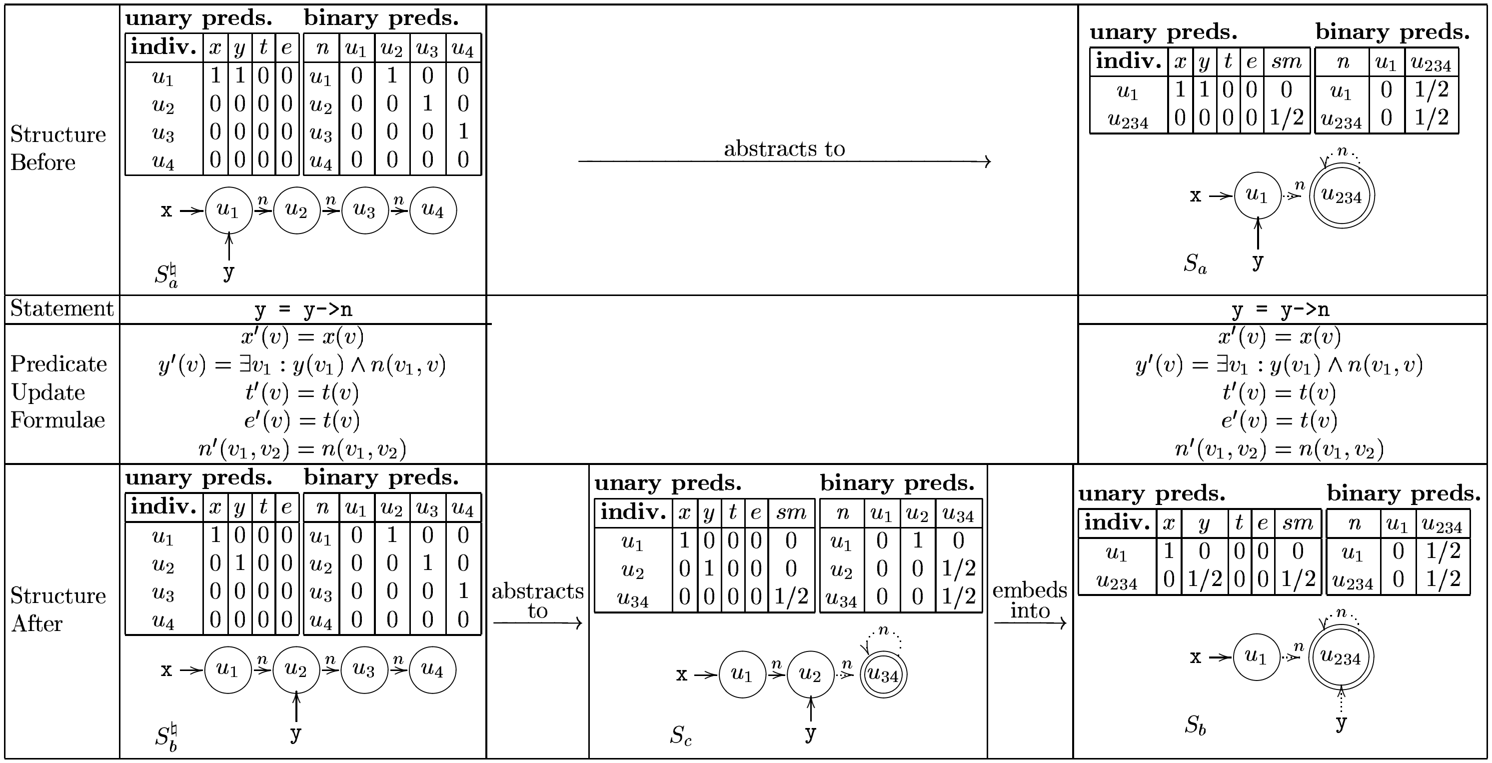
—By defining appropriate instrumentation predicates, it is possible to emulate some previous shape-analysis algorithms (e.g., [Chase et al. 1990; Jones and Muchnick 1981; Larus and Hilfinger 1988; Horwitz et al. 1989]).

Unfortunately, there is also bad news: the method described above and illustrated in Fig. 9 can be very imprecise. For instance, the statement `y = y->n` illustrated in Fig. 9 sets `y` to the value of `y->n`; i.e., it makes `y` point to the next element in the list. In the abstract semantics, the evaluation in structure S_a of the predicate-update formula $y'(v) = \exists v_1 : y(v_1) \wedge n(v_1, v)$ causes $\iota^{S_b}(y)(u_{234})$ to be set to 1/2: when

---

[3] The abstraction of $S_b^\natural$, as described in Sect. 2.5, is S_c. Fig. 9 illustrates that in the abstract semantics we also work with structures that are even further "blurred". We say that S_c *embeds into* S_b: u_1 in S_c maps to u_1 in S_b; u_2 and u_{34} in S_c both map to u_{234} in S_b; the n predicate of S_b is the "truth-blurring quotient" of n in S_c under this mapping.

Our notion of the 2-valued structures that a 3-valued structure represents will actually be based on the more general notion of embedding, rather than on "truth-blurring quotient" (cf. Defn. 4.8). Note that in Fig. 5, S_2 can be embedded into S_3; thus, structure S_3 also represents the acyclic lists of length 2 that are pointed to by `x`.



Structure Before, $S^\natural_a$:

unary preds.

indiv.	x	y	t	e
u_1	1	1	0	0
u_2	0	0	0	0
u_3	0	0	0	0
u_4	0	0	0	0

binary preds.

n	u_1	u_2	u_3	u_4
u_1	0	1	0	0
u_2	0	0	1	0
u_3	0	0	0	1
u_4	0	0	0	0

S_a:

unary preds.

indiv.	x	y	t	e	sm
u_1	1	1	0	0	0
u_{234}	0	0	0	0	1/2

binary preds.

n	u_1	u_{234}
u_1	0	1/2
u_{234}	0	1/2

Statement: `y = y->n`

Predicate Update Formulae:

$$x'(v) = x(v)$$
$$y'(v) = \exists v_1 : y(v_1) \wedge n(v_1, v)$$
$$t'(v) = t(v)$$
$$e'(v) = t(v)$$
$$n'(v_1, v_2) = n(v_1, v_2)$$

Structure After, $S^\natural_b$:

unary preds.

indiv.	x	y	t	e
u_1	1	0	0	0
u_2	0	1	0	0
u_3	0	0	0	0
u_4	0	0	0	0

binary preds.

n	u_1	u_2	u_3	u_4
u_1	0	1	0	0
u_2	0	0	1	0
u_3	0	0	0	1
u_4	0	0	0	0

S_c:

unary preds.

indiv.	x	y	t	e	sm
u_1	1	0	0	0	0
u_2	0	1	0	0	0
u_{34}	0	0	0	0	1/2

binary preds.

n	u_1	u_2	u_{34}
u_1	0	1	0
u_2	0	0	1/2
u_{34}	0	0	1/2

S_b:

unary preds.

indiv.	x	y	t	e	sm
u_1	1	0	0	0	0
u_{234}	0	1/2	0	0	1/2

binary preds.

n	u_1	u_{234}
u_1	0	1/2
u_{234}	0	1/2

Fig. 9. Commutative diagram that illustrates the relationship between (i) the transformation on 2-valued structures (defined by predicate-update formulae) that represents the concrete semantics for `y = y->n`, (ii) abstraction, and (iii) the transformation on 3-valued structures (defined by the same predicate-update formulae) that represents the simple abstract semantics for `y = y->n` obtained via the Reinterpretation Principle (Obs. 2.9).

$\exists v_1 : y(v_1) \wedge n(v_1, v)$ is evaluated in S_a, we have $\iota^{S_a}(y)(u_1) \wedge \iota^{S_a}(n)(u_1, u_{234}) = 1 \wedge 1/2 = 1/2$. Consequently, all we can surmise after the execution of `y = y->n` is that `y` may point to one of the heap cells that summary node u_{234} represents (see S_b). (This provides insight into where the algorithm of [Chase et al. 1990] loses precision.)

In contrast, the truth-blurring embedding of $S_b^{\natural}$ is S_c; thus col. 4 and row 4 of Fig. 9 show that the abstract semantics obtained via Obs 2.9 can lead to a structure that is not as precise as the abstract domain is capable of representing (cf. structures S_c and S_b). This observation motivates the mechanisms that are introduced in Sect. 6, where we define an improved abstract semantics. In particular, the mechanisms introduced in Sect. 6, are able to "materialize" new non-summary nodes from summary nodes as data structures are traversed. (Thus, Sect. 6 generalizes the algorithms of [Plevyak et al. 1993] and [Sagiv et al. 1998].) As we will see, this allows us to determine more precise shape descriptors for the data structures that arise, for example, in the `insert` program. In general, these techniques are important for retaining precision during the analysis of programs that, like `insert`, traverse linked data structures and perform destructive updating.

Because the mechanisms described in Sect. 6 are semantic reductions [Cousot and Cousot 1979], and because the Reinterpretation Principle falls out directly from the Embedding Theorem (Theorem 4.9), the correctness argument for the shape-analysis framework is surprisingly simple. (The reader is invited to compare the proof of Theorem 6.29 to that of Theorem 5.3.6 from [Sagiv et al. 1998].)

3. EXPRESSING THE CONCRETE SEMANTICS USING LOGIC

In this section, we define a meta-language for expressing the concrete operational semantics of programs (and programming languages), and use it to define a concrete collecting semantics for a simple programming language. The meta-language is based on first-order logic: each observable property is expressed via a predicate; the effect of every statement on every predicate's interpretation is given by means of a formula. The shape-analysis algorithm is generated from such a specification.

The rest of this section is organized as follows. Sect. 3.1 introduces the syntax of formulae for a first-order logic with transitive closure; the semantics of this logic is defined in Sect. 3.2. In Sect. 3.3, the logic is used to define a concrete operational semantics for statements and conditions of a C-like language (in particular, with heap-allocated storage and destructive updating through pointers). Finally, Sect. 3.4 presents a concrete collecting semantics, which associates a (potentially infinite) set of logical structures with every program point. (In subsequent sections of the paper, algorithms are developed that compute safe approximations to the collecting semantics.)

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 the logic.[4] We write first-order formulae over $\mathcal{P}$ using the logical connectives $\wedge$, $\vee$, $\neg$, and the quantifiers $\forall$ and

---

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

$\exists$. The symbol '=' denotes the equality predicate. The operator 'TC' denotes transitive closure on formulae.

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

DEFINITION 3.1. *A* **formula** *over the* **vocabulary** $\mathcal{P} = \{p_1, \ldots, p_n\}$ *is defined inductively, as follows:*

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

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

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

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_1 *is a formula with free variables* $\{v_1, v_2, \ldots, v_k\}$*, then* $(\exists v_1 : \varphi_1)$ *and* $(\forall v_1 : \varphi_1)$ *are both formulae with free variables* $\{v_2, v_3, \ldots, v_k\}$.

Transitive Closure. If φ_1 *is a formula with free variables* V *such that* $v_3, v_4 \notin V$*, then* $(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)$ *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.*

We also use several shorthand notations: $\varphi_1 \Rightarrow \varphi_2$ 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 $\neg(v_1 = v_2)$. For a binary predicate p, $p^+(v_3, v_4)$ is a shorthand for $(TC\ v_1, v_2 : p(v_1, v_2))(v_3, v_4)$. Finally, we make use of conditional expressions:

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

Table I lists the predicates used for representing the stores manipulated by programs that use the `List` data-type declaration from Fig. 1(a). In the general case, a program may use a number of different struct types. The vocabulary is then defined as follows:

$$\mathcal{P} \stackrel{\text{def}}{=} \{x \mid \mathtt{x} \in PVar\} \cup \{sel \mid \mathtt{sel} \in Sel\}, \tag{3}$$

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

3.2 Semantics of First-Order Logic

In this section, we define the (2-valued) semantics for first-order logic with transitive closure in the standard way.

DEFINITION 3.2. *A* **2-valued interpretation** *of the language of formulae over* $\mathcal{P}$ *is a* **2-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:*

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

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 **(2-valued) meaning** *of a formula φ, denoted by $[\![\varphi]\!]_2^S(Z)$, yields a truth value in $\{0, 1\}$. The meaning of φ is defined inductively as follows:*

Atomic Formulae. For an atomic formula consisting of a logical literal $\boldsymbol{l} \in \{\mathbf{0}, \mathbf{1}\}$, $[\![\boldsymbol{l}]\!]_2^S(Z) = l$ (where $l \in \{0, 1\}$).

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

$$[\![p(v_1, \ldots, v_k)]\!]_2^S(Z) = \iota^S(p)(Z(v_1), \ldots, Z(v_k))$$

For an atomic formula of the form $(v_1 = v_2)$,[5]

$$[\![v_1 = v_2]\!]_2^S(Z) = \begin{cases} 0 & Z(v_1) \neq Z(v_2) \\ 1 & Z(v_1) = Z(v_2) \end{cases}$$

Logical Connectives. When φ is a formula built from subformulae φ_1 and φ_2,

$$\begin{aligned} [\![\varphi_1 \wedge \varphi_2]\!]_2^S(Z) &= \min([\![\varphi_1]\!]_2^S(Z), [\![\varphi_2]\!]_2^S(Z)) \\ [\![\varphi_1 \vee \varphi_2]\!]_2^S(Z) &= \max([\![\varphi_1]\!]_2^S(Z), [\![\varphi_2]\!]_2^S(Z)) \\ [\![\neg\varphi_1]\!]_2^S(Z) &= 1 - [\![\varphi_1]\!]_2^S(Z) \end{aligned}$$

Quantifiers. When φ is a formula that has a quantifier as the outermost operator,

$$\begin{aligned} [\![\forall v_1 : \varphi_1]\!]_2^S(Z) &= \min_{u \in U^S} [\![\varphi_1]\!]_2^S(Z[v_1 \mapsto u]) \\ [\![\exists v_1 : \varphi_1]\!]_2^S(Z) &= \max_{u \in U^S} [\![\varphi_1]\!]_2^S(Z[v_1 \mapsto u]) \end{aligned}$$

Transitive Closure. When φ is a formula of the form $(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)$,

$$[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_2^S(Z) =$$

$$\max_{\substack{n \geq 1, u_1, \ldots, u_{n+1} \in U, \\ Z(v_3) = u_1, Z(v_4) = u_{n+1}}} \min_{i=1}^{n} [\![\varphi_1]\!]_2^S(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

We say that S and Z **satisfy** *φ (denoted by $S, Z \models \varphi$) if $[\![\varphi]\!]_3^S(Z) = 1$. We write $S \models \varphi$ if for every Z we have $S, Z \models \varphi$.*

In the sequel, we denote the set of 2-valued structures by 2-STRUCT$[\mathcal{P}]$.

As already discussed in Sect. 2.2, logical structures are used to encode stores as follows: Individuals represent memory locations in the heap; pointers from the stack into the heap are represented by unary "pointed-to-by-variable-q" predicates; and pointer-valued fields of data structures are represented by binary predicates.

Notice that Defns. 3.1 and 3.2 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 follow this route.

---

[5] 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.

st	φ_p^{st}
`x = NULL`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} \mathbf{0}$
`x = t`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} t(v)$
`x = t->sel`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} \exists v_1 : t(v_1) \wedge sel(v_1, v)$
`x->sel = NULL`	$\varphi_{sel}^{st}(v_1, v_2) \stackrel{\text{def}}{=} sel(v_1, v_2) \wedge \neg x(v_1)$
`x->sel = t` (assuming that `x->sel == NULL`)	$\varphi_{sel}^{st}(v_1, v_2) \stackrel{\text{def}}{=} sel(v_1, v_2) \vee (x(v_1) \wedge t(v_2))$
`x = malloc()`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} isNew(v)$ $\varphi_z^{st}(v) \stackrel{\text{def}}{=} \quad z(v) \wedge \neg isNew(v)$, for each $\mathtt{z} \in (PVar - \{\mathtt{x}\})$ $\varphi_{sel}^{st}(v_1, v_2) \stackrel{\text{def}}{=} sel(v_1, v_2) \wedge \neg isNew(v_1) \wedge \neg isNew(v_2)$ for each $\mathtt{sel} \in PSel$

Table III. Predicate-update formulae that define the semantics of statements that manipulate pointers and pointer-valued fields.

3.3 The Meaning of Program Statements

For every statement st, the new values of every predicate p are defined via a predicate-update formula φ_p^{st}.

Definition 3.3. *Let st be a program statement, and for every arity-k predicate p in vocabulary $\mathcal{P}$, let φ_p^{st} be the formula over free variables $v_1, \ldots, v_k$ that defines the new value of p after st. Then the $\mathcal{P}$* **transformer associated with** *st, denoted by $[\![st]\!]$: 2-STRUCT$[\mathcal{P}] \to$ 2-STRUCT$[\mathcal{P}]$, is defined as follows:*

$$[\![st]\!](S) = \langle U^S, \lambda p.\lambda u_1, \ldots, u_k.[\![\varphi_p^{st}]\!]_2^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k])\rangle$$

In the sequel, we avoid cluttering the definition of statement transformers by omitting predicate-update formulae for predicates whose value is not changed by the statement, i.e., for predicates whose predicate-update formula φ_p^{st} is merely $p(v_1, v_2, \ldots, v_k)$.

Example 3.4. Table III lists the predicate-update formulae that define the operational semantics of the five kinds of statements that manipulate C structures.

In Table III, and also in later tables, we simplify the presentation of the semantics by breaking the statement `x->sel = t` into two parts: (i) `x->sel = NULL`, and (ii) `x->sel = t`, assuming that `x->sel == NULL`.

Defn. 3.3 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 storage-allocation statements we need to use the modified definition of $[\![st]\!](S)$ given in Defn. 3.5, which first allocates a new individual u_{new}, and then invokes predicate-update formulae in a manner similar to Defn. 3.3.

Definition 3.5. *Let $st \equiv \mathtt{x} = \mathtt{malloc}()$ and let $isNew \notin \mathcal{P}$ be a unary predicate. For every $p \in \mathcal{P}$, let φ_p^{st} be a predicate-update formula over the vocabulary $\mathcal{P} \cup \{isNew\}$. Then the $\mathcal{P}$* **transformer associated with** *$st \equiv \mathtt{x} = \mathtt{malloc}()$, denoted*

w	$cond(w)$
`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`	$\exists v : x(v)$

Table IV. Formulae for four kinds of primitive conditions involving pointer variables.

by $[\![\mathtt{x} = \mathtt{malloc()}]\!]$, *is defined as follows:*

$$
\begin{array}{l}
[\![\mathtt{x} = \mathtt{malloc()}]\!](S) = \\
\quad \textbf{let } U' = U^S \cup \{u_{new}\},\ \textit{where } u_{new} \textit{ is an individual not in } U^S \\
\quad \textbf{and } \iota' = \lambda p \in (\mathcal{P} \cup \{\mathit{isNew}\}).\lambda u_1, \ldots, u_k. \\
\qquad\qquad \begin{cases} 1 & p = \mathit{isNew} \textit{ and } u_1 = u_{new} \\ 0 & p = \mathit{isNew} \textit{ and } u_1 \neq u_{new} \\ 0 & \begin{array}{l} p \neq \mathit{isNew} \textit{ and there exists } i, \\ 1 \leq i \leq k,\ \textit{such that } u_i = u_{new} \end{array} \\ \iota^S(p)(u_1, \ldots, u_k) & \text{otherwise} \end{cases} \\
\quad \textbf{in } \langle U', \lambda p \in \mathcal{P}.\lambda u_1, \ldots, u_k.[\![\varphi_p^{st}]\!]_2^{\langle U', \iota' \rangle}([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k])\rangle
\end{array}
$$

In Defn. 3.5, ι' is created from ι as follows: (i) $\mathit{isNew}(u_{new})$ is set to 1, (ii) $\mathit{isNew}(u_1)$ is set to 0 for all other individuals $u_1 \neq u_{new}$, and (iii) all predicates are set to 0 when one or more arguments is u_{new}. The predicate-update operation in Defn. 3.5 is very similar to the one in Defn. 3.3 after ι' has been set. (Note that the p in "$\iota' = \lambda p. \ldots$" ranges over $\mathcal{P} \cup \{\mathit{isNew}\}$, whereas the p in "$\lambda p. \ldots$" appearing in the last line of Defn. 3.5 ranges over $\mathcal{P}$.)

2-valued formulae also provide a way to define the meaning of program conditions. 2-valued (closed) formulae for four kinds of primitive program conditions that involve pointer variables are shown in Table IV; the formula to express the meaning of a compound program condition involving pointer variables would have the formulae from Table IV as constituents. To keep things simple, we do not use examples in which program conditions have side-effects; however, it should be noted that it is possible to handle side-effects in program conditions in the same way that it is done for statements, namely, by providing appropriate predicate-update formulae.

Finally, it should also be noted that the concrete semantics that has been defined is already somewhat abstract.[6] By design, the concrete semantics ignores a number of details:

- The only parts of the store that the concrete semantics keeps track of are the pointer variables and the cells of heap-allocated storage.
- The concrete semantics does not track changes to stores caused by assignment statements that perform actions other than pointer manipulations (e.g., arithmetic, etc.).

---

[6] This is an approach that has also been used in previous work on shape-analysis [Sagiv et al. 1998].

—The concrete semantics is assumed to “go both ways” at a branch point in the control-flow graph where the program condition involves something other than pointer-valued quantities (see Sect. 3.4).

Such assumptions build a small amount of abstraction into the “concrete” semantics. The consequence of these assumptions is that the collecting semantics defined in Sect. 3.4 may associate a control-flow-graph vertex with more concrete stores (i.e., 2-valued structures) than would be the case had we started with a conventional concrete semantics.

3.4 Collecting Semantics

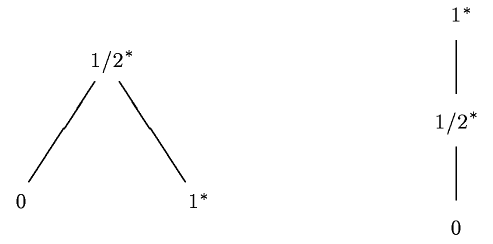
We now turn to the collecting semantics. For each vertex v of control-flow graph G, the set $\mathit{ConcStructSet}[v]$ is a (potentially infinite) set of structures that may arise on entry to v for some potential input. For our purposes, it is convenient to define $\mathit{ConcStructSet}[v]$ as the least fixed point (in terms of set inclusion) of the following system of equations (over the variables $\mathit{ConcStructSet}[v]$):

$$\mathit{ConcStructSet}[v] = \begin{cases} \{\langle\emptyset,\emptyset\rangle\} & \text{if } v = \mathit{start} \\ \left.\begin{array}{ll} & \displaystyle\bigcup_{\substack{w \to v \,\in\, E(G),\\ w \,\in\, As(G)}} \{[\![st(w)]\!](S) \mid S \in \mathit{ConcStructSet}[w]\} \\ \cup & \displaystyle\bigcup_{\substack{w \to v \,\in\, E(G),\\ w \,\in\, Id(G)}} \{S \mid S \in \mathit{ConcStructSet}[w]\} \\ \cup & \displaystyle\bigcup_{w \to v \in Tb(G)} \{S \mid S \in \mathit{ConcStructSet}[w] \text{ and } S \models \mathit{cond}(w)\} \\ \cup & \displaystyle\bigcup_{w \to v \in Fb(G)} \{S \mid S \in \mathit{ConcStructSet}[w] \text{ and } S \models \neg\mathit{cond}(w)\} \end{array}\right\} & \text{otherwise} \end{cases} \tag{4}$$

In equation (4), $As(G)$ denotes the set of assignment statements that manipulate pointers; $Id(G)$ denotes the set of assignment statements that perform actions other than pointer manipulations, plus the branch points for which the program condition involves something other than just pointer-valued quantities (in both cases, these control-flow graph vertices are uninterpreted); $Tb(G) \subseteq E(G)$ and $Fb(G) \subseteq E(G)$ are the subsets of G’s edges that represent the true and false branches, respectively, from branch points that involve only pointer-valued quantities ($\mathit{cond}(w)$ denotes the formula for the program condition at w). (An edge whose source is an `assert` statement that involves only pointer-valued quantities would be handled just like a true-branch edge.)

4. REPRESENTING SETS OF STORES USING 3-VALUED LOGIC

In this section, we show how 3-valued logical structures can be used to conservatively represent sets of concrete stores. Sect. 4.1 defines 3-valued logic. Sect. 4.2 introduces the concept of *embedding*, which is used to relate concrete (2-valued) and abstract (3-valued) structures. In particular, Sect. 4.2 contains the Embedding Theorem, which is the main tool for conservative extraction of store properties (cf. Sect. 2.6). The lattice of static information that will be used in Sect. 6 has



(a) Information order (b) Logical order

Fig. 10. The semi-bilattice of 3-valued logic. (The symbol $*$ attached to 1/2 and 1 indicates that these are the "designated values", which correspond to "potential truth".)

as its elements sets of 3-valued structures (ordered by set inclusion). To guarantee that the analysis terminates when applied to a program that contains a loop, we need a way to ensure that the number of 3-valued structures that can arise is finite. For this reason, in Sect. 4.3 we introduce the set of bounded structures, and show how every 3-valued structure can be mapped into a bounded structure.

(Sect. 5 introduces an additional mechanism for refining the abstractions discussed in the present section.)

4.1 Kleene's 3-Valued Semantics

In this section, we define Kleene's 3-valued semantics for first-order logic with transitive closure. We say that the values 0 and 1 are *definite values* and that 1/2 is an *indefinite value*, and define a partial order $\sqsubseteq$ on truth values to reflect information content: $l_1 \sqsubseteq l_2$ denotes that l_1 has more definite information than l_2:

DEFINITION 4.1. [**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 3-valued semantics of logic is monotonic in the information order (see Table II and Defn. 4.2).

As shown in Fig. 10, the values 0, 1, and 1/2 form a mathematical structure known as a semi-bilattice (see [Ginsberg 1988]). A semi-bilattice has two orderings: the *information order* and the *logical order*:

—The information order is the one defined in Defn. 4.1, which captures "(un)certainty".

—The logical order is the one used in Table II: 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.).

A value that is "far enough up" in the logical order indicates "potential truth", and is called a *designated value*. In Fig. 10, 1/2 and 1 are the designated values. This means that a structure S potentially satisfies a formula when the formula's interpretation with respect to S is either 1/2 or 1 (see Defn. 4.2).

We now generalize Defn. 3.2 to define the meaning of a formula with respect to a 3-valued structure. The generalized definition assumes that every 3-valued structure includes a unary predicate sm, which will be used to define the meaning of the syntactic equality symbol ('$=$'). As explained earlier, sm formalizes the notion of "summary nodes" (i.e., individuals of a 3-valued structure that may represent more than one individual from corresponding 2-valued structures).

DEFINITION 4.2. *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:*

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

For an assignment Z*, the* **(3-valued) meaning** *of a formula* φ*, denoted by* $[\![\varphi]\!]^S_3(Z)$*, now yields a truth value in* $\{0, 1, 1/2\}$*. The meaning of* φ *is defined inductively as in Defn. 3.2, with the following changes:*

Atomic Formulae. For an atomic formula of the form $(v_1 = v_2)$*,*

$$[\![v_1 = v_2]\!]^S_3(Z) = \begin{cases} 0 & Z(v_1) \neq Z(v_2) \\ 1 & Z(v_1) = Z(v_2) \textit{ and } \iota^S(sm)(Z(v_1)) = 0 \\ 1/2 & \text{otherwise} \end{cases} \tag{5}$$

We say that S *and* Z **potentially satisfy** φ*, denoted by* $S, Z \models_3 \varphi$*, if* $[\![\varphi]\!]^S_3(Z) = 1/2$ *or* $[\![\varphi]\!]^S_3(Z) = 1$*. We write* $S \models_3 \varphi$ *if for every* Z *we have* $S, Z \models_3 \varphi$*.*

In the sequel, we denote the set of 3-valued structures by $\mathit{3\text{-}STRUCT}[\mathcal{P} \cup \{sm\}]$*.*

In Defn. 4.2, the meaning of a formula of the form $v_1 = v_2$ is defined in terms of the sm predicate and the "identically-equal" relation on individuals (denoted by the symbol '$=$'):

- Non-identical individuals u_1 and u_2 are unequal (i.e., if $u_1 \neq u_2$, then $[\![v_1 = v_2]\!]^S_3([v_1 \mapsto u_1, v_2 \mapsto u_2])$ evaluates to 0).
- A non-summary individual must be equal to itself (i.e., if $sm(u) = 0$, then $[\![v_1 = v_2]\!]^S_3([v_1 \mapsto u, v_2 \mapsto u])$ evaluates to 1).
- In all other cases, we throw up our hands and return 1/2.

EXAMPLE 4.3. Consider the structure S_3 from Fig. 5 and formula (1),

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

which expresses the "is-shared" property. For the assignment $Z_1 = [v \mapsto u]$, we have

$$\begin{aligned} [\![\varphi_{is}]\!]^{S_3}_3(Z_1) &= \max_{u', u'' \in \{u_1, u\}} [\![n(v_1, v) \land n(v_2, v) \land v_1 \neq v_2]\!]^{S_3}_3([v \mapsto u, v_1 \mapsto u', v_2 \mapsto u'']) \\ &= 1/2, \end{aligned} \tag{6}$$

and thus $S_3, Z_1 \models_3 \varphi_{is}$. In contrast, for the assignment $Z_2 = [v \mapsto u_1]$, we have

$$\begin{aligned} [\![\varphi_{is}]\!]^{S_3}_3(Z_2) &= \max_{u', u'' \in \{u_1, u\}} [\![n(v_1, v) \land n(v_2, v) \land v_1 \neq v_2]\!]^{S_3}_3([v \mapsto u_1, v_1 \mapsto u', v_2 \mapsto u'']) \\ &= 0, \end{aligned}$$

and thus $S_3, Z_2 \not\models_3 \varphi_{is}$.

3-valued logic retains a number of properties that are familiar from 2-valued logic, such as De Morgan's laws, associativity of $\wedge$ and $\vee$, and distributivity of $\wedge$ over $\vee$ (and vice versa).

Kleene's semantics is monotonic in the information order:

Lemma 4.4. *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,*

$$[\![\varphi]\!]_3^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S'}(Z). \tag{7}$$

4.2 Embedding into 3-Valued Structures

In this section, we introduce the concept of *embedding*, which provides a way to relate 2-valued and 3-valued structures and formulate the Embedding Theorem, which relates 2-valued and 3-valued interpretations of a given formula.

Convention. To avoid the need to work with different vocabularies at the concrete and abstract levels, we assume that the sm predicate is defined in every concrete 2-valued structure, where it has the trivial fixed meaning of 0 for all individuals. In the concrete operational semantics, we assume that sm is set to 0 for the individual allocated by `x = malloc()`, and is never changed by any of the other kinds of statements; thus, in the concrete operational semantics, for each non-`malloc` statement st, the predicate-update formula for sm is always the trivial one: $\varphi_{sm}^{st}(v) = sm(v)$. $\square$

4.2.1 *Embedding Order.* We define the *embedding ordering* on structures as follows:

Definition 4.5. *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 surjective function. We say that f* **embeds** *S in S' (denoted by $S \sqsubseteq^f S'$) if (i) for every predicate symbol $p \in \mathcal{P} \cup \{sm\}$ of arity k and all $u_1, \ldots, u_k \in U^S$,*

$$\iota^S(p)(u_1, \ldots, u_k) \sqsubseteq \iota^{S'}(p)(f(u_1), \ldots, f(u_k)) \tag{8}$$

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

$$(|\{u \mid f(u) = u'\}| > 1) \sqsubseteq \iota^{S'}(sm)(u') \tag{9}$$

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 (8) applies to sm, as well; therefore, $\iota^{S'}(sm)(u')$ can never be 1.

4.2.2 *Tight Embedding.* A *tight embedding* is a special kind of embedding—one in which information loss is minimized when multiple individuals of S are mapped to the same individual in S':

Definition 4.6. *A structure $S' = \langle U^{S'}, \iota^{S'} \rangle$ is a* **tight embedding** *of $S = \langle U^S, \iota^S \rangle$ if there exists a surjective function $t\_embed \colon U^S \to U^{S'}$ such that, for every $p \in \mathcal{P}$ of arity k,*

$$\iota^{S'}(p)(u'_1, \ldots, u'_k) = \bigsqcup_{t\_embed(u_i) = u'_i, 1 \le i \le k} \iota^S(p)(u_1, \ldots, u_k) \tag{10}$$

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

$$\iota^{S'}(sm)(u') = (|\{u | t\_embed(u) = u'\}| > 1) \sqcup \bigsqcup_{t\_embed(u)=u'} \iota^{S}(sm)(u) \tag{11}$$

When a surjective function t_embed possesses both properties (10) and (11), we say that $S' = t\_embed(S)$.

It is immediately apparent from Defn. 4.6 that the tight embedding of a structure S by a function t_embed embeds S in $t\_embed(S)$, i.e., $S \sqsubseteq^{t\text{-}embed} t\_embed(S)$.

It is also apparent from Defn. 4.6 how several individuals from U^S can "lose their identity" by being mapped to the same individual in $U^{S'}$:

EXAMPLE 4.7. 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, by (5), $[\![v_1 = v_2]\!]_3^{S'}([v_1 \mapsto u', v_2 \mapsto u']) = 1/2$.

In addition to defining what it means for a 2-valued structure to be embedded in a 3-valued structure, Defns. 4.5 and 4.6 also define what it means for a 3-valued structure to be embedded in a 3-valued structure. Eqns. (9) and (11) have the form given above so that $\sqsubseteq$ is transitive and 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).

4.2.3 *Concretization of* 3*-Valued Structures.* Embedding also allows us to define the (potentially infinite) set of concrete structures that a single 3-valued structure represents:

DEFINITION 4.8. **(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.,*

$$\gamma(S) = \{S^{\natural} \in \textit{2-STRUCT}[\mathcal{P}] \mid S^{\natural} \sqsubseteq S\}. \tag{12}$$

4.2.4 *The Embedding Theorem.* Informally, the Embedding Theorem 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.

To formalize this, we extend mappings on individuals to operate on assignments: 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 $(f \circ Z)(v) = f(Z(v))$.

The formal statement of the Embedding Theorem is as follows:

THEOREM 4.9. [**Embedding Theorem**]. *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: Appears in Appendix B.

Note that if S is a 2-valued structure, then we have $[\![\varphi]\!]_2^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S'}(f \circ Z)$.

EXAMPLE 4.10. Continuing Example 4.7, we can illustrate the Embedding Theorem on the formula $\varphi \equiv v_1 = v_2$ and the embedding $f \equiv t\_embed$, as follows:

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

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

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 4.4).

The use of this machinery provides several advantages for program analysis:

- The Embedding Theorem provides a systematic way to use an abstract (3-valued) structure S to answer questions about properties of the concrete (2-valued) structures that S represents. It ensures that it is safe to evaluate a formula φ on a single 3-valued structure S, instead of evaluating φ in all structures $S^\natural$ that are members of the (potentially infinite) set $\gamma(S)$. In particular, a definite value for φ in S means that φ yields the same definite value in all $S^\natural \in \gamma(S)$.
- The Embedding Theorem allows us to extract information from *either* the concrete world or the abstract world via the *same* formula—the same syntactic expression can be interpreted either in the 2-valued world or the 3-valued world; the consistency of the information obtained is ensured by the Embedding Theorem.

4.2.5 *"Summary Nodes" and Equality.* Because predicate *sm* receives special treatment in Defns. 4.5 and 4.6, the definitions of embedding and tight embedding look a bit awkward. It would be possible to sidestep this by assuming that every structure—2-valued or 3-valued—includes a binary predicate *eq* (rather than a unary predicate *sm*); *eq* is then used to define the meaning of the syntactic equality symbol ('='). In 2-valued structures, *eq* merely represents the "identically-equal" relation on individuals:

$$\iota^S(eq)(v_1, v_2) = (v_1 = v_2).$$

In embeddings, the status of *eq* is no different from the other predicates: its value must abide by Eqn. (8) (or Eqn. (10), in the case of a tight embedding). In both 2-

valued and 3-valued structures, the meaning of the syntactic equality symbol ('=') is defined by

$$[\![v_1 = v_2]\!]^S(Z) = \iota^S(eq)(v_1, v_2).$$

With this approach, sm can be defined as an instrumentation predicate:

$$sm(v) \stackrel{\text{def}}{=} (v \neq v).$$

It is then a consequence of Defn. 3.2 that sm always evaluates to 0 in a 2-valued structure. In 3-valued structures created via embedding, Eqns. (5) and (9) (as well as Eqn. (11), in the case of a tight embedding) follow from the surjectivity of the embedding function and Eqn. (8) (Eqn. (10), in the case of a tight embedding).

One motivation for introducing sm explicitly was to resemble more closely the shape-analysis algorithms presented in earlier work, which have an explicit notion of "summary nodes" [Jones and Muchnick 1981; Chase et al. 1990; Sagiv et al. 1998; Wang 1994].

A second motivation was provided by the fact that the presence of an explicit sm predicate reduces the amount of space needed to represent 3-valued structures. In 3-valued structures, semantic equality no longer coincides with the "identically-equal" relation on individuals (cf. Examples 4.7 and 4.10)—hence, some storage must be devoted to representing semantic equality. The unary predicate sm can be represented in space linear in the number of individuals, as opposed to the binary predicate eq, which takes quadratic space.

4.3 Bounded Structures

To guarantee that shape analysis terminates for a program that contains a loop, we require that the number of potential structures for a given program be finite.[7] Toward this end, we make the following definition:

DEFINITION 4.11. *A* **bounded structure** *over vocabulary $\mathcal{P} \cup \{sm\}$ 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$[\mathcal{P} \cup \{sm\}]$ denotes the set of such structures.

We will use the symbol $\mathcal{A}$ to denote the set of unary predicate symbols of vocabulary $\mathcal{P}$. The consequence of Defn. 4.11 is that there is an upper bound on the size of structures $S \in$ B-STRUCT$[\mathcal{P} \cup \{sm\}]$, i.e., $|U^S| \leq 3^{|\mathcal{A}|}$.

EXAMPLE 4.12. Consider the class of bounded structures associated with the `List` data-type declaration from Fig. 1(a). Here the predicate symbols are $\mathcal{P} = \{n\} \cup \{x \mid \texttt{x} \in PVar\}$. For the `insert` program from Fig. 1(b), the program variables are `x`, `y`, `t` and `e`, yielding unary predicates x, y, t, and e. Therefore, the maximal number of individuals in a structure is $3^4 = 81$. (However, this is a worst-case bound; an application of the analysis does not necessarily create structures that have this many individuals. For instance, at most 6 individuals arise in any structure in the complete analysis of `insert`.)

---

[7] An alternative would be to define widening operators that guarantee termination [Cousot and Cousot 1979].

4.3.1 *Canonical Abstraction.*

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

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

The name "$u_{\{p \in \mathcal{A} \mid \iota^S(p)(u)=1\},\{p \in \mathcal{A} \mid \iota^S(p)(u)=0\}}$" is known as the **canonical name** of individual u. The subscript on the canonical name of u involves two sets of unary predicate symbols: (i) those that are true at u, and (ii) those that are false at u.

Henceforth, we assume in our examples that $\mathcal{A}$, the set of unary predicates, is the set $\{x, y, t, e, is, c_n, r_{x,n}, r_{y,n}, r_{t,n}, r_{e,n}\}$.

EXAMPLE 4.14. In structure S_{reach} from Fig. 8, the canonical names of the four individuals are as follows:

Individual	**Canonical Name**
u_1	$u_{\{x,r_{x,n}\},\{y,t,e,is,c_n,r_{y,n},r_{t,n},r_{e,n}\}}$
u	$u_{\{r_{x,n}\},\{x,y,t,e,is,c_n,r_{y,n},r_{t,n},r_{e,n}\}}$
u_2	$u_{\{y,r_{x,n},r_{y,n}\},\{x,t,e,is,c_n,r_{t,n},r_{e,n}\}}$
u'	$u_{\{r_{x,n},r_{y,n}\},\{x,y,t,e,is,c_n,r_{t,n},r_{e,n}\}}$

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., $t\_embed_c(t\_embed_c(S)) = t\_embed_c(S)$).

For any two bounded structures $S, S' \in$ B-STRUCT$[\mathcal{P} \cup \{sm\}]$, 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 $p \in \mathcal{P} \cup \{sm\}$, check that the predicates $\iota^S(p)$ and $\iota^{S'}(p)$ are equal.

It is straightforward to generalize Defn. 4.13 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.

4.3.2 *Relationship of Canonical Abstraction to Previous Work.*

Canonical abstraction is a generalization of the abstraction functions that have been used in some of the previous work on shape analysis [Jones and Muchnick 1981; Chase et al. 1990; Wang 1994; Sagiv et al. 1998].[8] For instance, the abstraction predicates used in [Sagiv et al. 1998] are the "pointed-to-by-variable-x" predicates and thus correspond to the instantiation of canonical abstraction discussed in Example 4.14. Earlier, Jones and Muchnick proposed making even finer distinctions by keeping

---

[8] The shape-analysis algorithms presented in [Jones and Muchnick 1981; Chase et al. 1990; Wang 1994; Sagiv et al. 1998] are described in terms of various kinds of *Storage Shape Graphs* (SSGs), not bounded structures. Our comparison is couched in terms of the terminology of the present paper.

exact information on elements within a distance k from a variable [Jones and Muchnick 1981]. In [Wang 1994], in addition to "pointed-to-by-variable-x" predicates, there are predicates of the form "was-pointed-to-by-variable-x-at-program-point-p".

Defn. 4.13 generalizes these ideas to define a set of bounded structures in terms of any fixed set of unary "abstraction properties" on individuals:

OBSERVATION 4.15. [**Abstraction Principle**]. *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* S *in which each individual of* S *represents an equivalence class of individuals from* $S^\natural$*. This method of collapsing structures always yields bounded structures.*

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 logical structures, it is possible 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.
- At various stages, we work with non-bounded structures, and return to bounded structures by applying t_embed_c (see Sect. 6). The Embedding Theorem ensures that the operations we apply to 3-valued structures are safe, even when we are working with non-bounded structures.

5. INSTRUMENTATION PREDICATES

It is possible to improve the precision of the analysis by using an abstract domain that makes finer distinctions among the concrete structures. As discussed in Sect. 2.6, the instrumentation principle is the main tool for achieving this—that is, the notion of which "finer distinctions" to make is defined using instrumentation predicates, which record information derived from other predicates.[9]

Formally, we assume that the set of predicates $\mathcal{P}$ is partitioned into two disjoint sets, the core predicates, denoted by $\mathcal{C}$, and the instrumentation predicates, denoted by $\mathcal{I}$. Furthermore, the meaning of every instrumentation predicate is defined in terms of a formula over the core predicates.[10]

EXAMPLE 5.1. Table V lists some examples of instrumentation predicates, and Table VI gives their defining formulae.

Instrumentation predicates can increase the precision of a program-analysis algorithm in at least three ways:

---

[9] In the literature on logic, these predicates are sometimes called *derived* predicates.

[10] For the sake of abbreviation, it is sometimes convenient to allow an instrumentation predicate's defining formula to be defined in terms of other instrumentation predicates; however, such defining formulae should not be mutually recursive.

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]
$r_{x,n}(v)$	Is v (transitively) reachable from pointer variable `x` along `n` fields?	separating disjoint data structures	[Sagiv et al. 1998]
$r_n(v)$	Is v reachable from some pointer variable along `n` fields (i.e., is v a non-garbage element)?	compile-time garbage collection	
$c_n(v)$	Is v on a directed cycle of `n` fields?	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 V. Examples of instrumentation predicates.

$$\varphi_{is}(v) \stackrel{\text{def}}{=} \exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2 \tag{13}$$

$$\varphi_{r_{x,n}}(v) \stackrel{\text{def}}{=} x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v) \tag{14}$$

$$\varphi_{r_n}(v) \stackrel{\text{def}}{=} \bigvee_{\mathtt{x} \in PVar} (x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v)) \tag{15}$$

$$\varphi_{c_n}(v) \stackrel{\text{def}}{=} n^+(v, v) \tag{16}$$

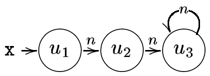
$$\varphi_{c_{f.b}}(v) \stackrel{\text{def}}{=} \forall v_1 : f(v, v_1) \Rightarrow b(v_1, v) \tag{17}$$

$$\varphi_{c_{b.f}}(v) \stackrel{\text{def}}{=} \forall v_1 : b(v, v_1) \Rightarrow f(v_1, v) \tag{18}$$

Table VI. Formulae for the instrumentation predicates listed in Table V.

(1) The value stored for an instrumentation predicate in a given 3-valued structure S may be more precise than that obtained by evaluating the instrumentation predicate's defining formula (Obs. 2.8). For example, in structure $S_{acyclic}$ from Fig. 6, $\iota^{S_{acyclic}}(c_n)(u) = 0$ despite the fact that φ_{c_n} evaluates to $1/2$ on u.

(2) The set of concrete structures that a given 3-valued structure S represents (as defined by Eqn. (12)) is, in general, decreased if we augment S with additional instrumentation predicates that have definite values for at least some combinations of individuals. For example, structure $S_{acyclic}$ from Fig. 6 is structure S_3 from Fig. 5 augmented with instrumentation predicate c_n, for which $c_n(u_1) = 0$ and $c_n(u) = 0$. Thus, $S_{acyclic}$ cannot possibly represent 2-valued structures with cyclic nodes; in contrast, among the structures that S_3 represents is the following concrete cyclic list:



(3) Unary instrumentation predicates that are used as abstraction predicates refine the set of bounded structures. For example, using the "is-shared" predicate is as an abstraction predicate leads to an algorithm that is more precise than the one given in [Sagiv et al. 1998]. The latter does not distinguish between shared and unshared individuals, and thus loses accuracy for stores that con-

tain a shared heap cell that is not directly pointed to by a program variable. Adopting *is* as an additional abstraction predicate improves the accuracy of shape analysis because concrete-store elements that are shared and concrete-store elements that are not shared are represented by abstract individuals that have different canonical names.

It is important to note that the instrumentation predicates do not have to be unary. Furthermore, not all of the unary instrumentation predicates need necessarily be used as abstraction predicates.

Adding more unary instrumentation predicates and using them as abstraction predicates increases the worst-case cost of the analysis, since the number of individuals in bounded structures is proportional to $3^{|\mathcal{A}|}$. However, our initial experience indicates that the opposite happens in practice; by using the "right" unary instrumentation predicates, the cost of the analysis can be significantly decreased (see Sect. 7.4).

5.1 Updating Instrumentation Predicates

Because each instrumentation predicate is defined by means of a formula over the core predicates (cf. Table VI), 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 explicitly. In particular, this is often the case for an instrumentation predicate's value for a summary node.

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}).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 is equivalent to merely reevaluating p's defining formula in the structure obtained after st has been executed. As demonstrated in Sect. 2, reevaluation may yield many indefinite values, and hence the trivial update formula is often unsatisfactory. It is preferable, therefore, to devise predicate-update formulae that minimize reevaluations of φ_p.

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

$$[\![\varphi_p^{st}]\!]_2^{S^\natural}(Z) = [\![\varphi_p]\!]_2^{[\![st]\!](S^\natural)}(Z). \tag{19}$$

In the above definition, $[\![st]\!](S^\natural)$ denotes a version of the operation defined in Defns. 3.3 and 3.5 in which $\mathcal{P}$ is restricted to $\mathcal{C}$. We make the assumption that the predicate-update formula for an instrumentation predicate is defined solely in

st	φ_{is}^{st}
`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 that `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 VII. Predicate-update formulae for the instrumentation predicate is.

terms of core predicates. An instrumentation predicate's formula can always be put in this form by repeated substitution until only core predicates remain.

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 formulae do maintain correct instrumentation; however, they may yield very imprecise answers when applied to 3-valued structures.

5.2 Updating Sharing

Table VII gives the predicate-update formulae for the instrumentation predicate is. (It lists formulae only for the statements that may affect the value of is.) The assignment to `x->n = NULL` can only change the value of is to 0, and only for an element pointed to by `x->n`. Therefore, in Table VII $\varphi_{is}[n \mapsto \varphi_n^{st}]$ is evaluated only for elements pointed to by `x->n`. Similarly, the assignment `x->n = t` (assuming that `x->n == NULL`) can only change the value of is to 1, and only for an element that is pointed to by `t` and 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.

5.3 Updating Reachability and Cyclicity

In this section, we discuss how to define predicate-update formulae, $\varphi_{r_{x,n}}^{st}(v)$, that maintain the correct instrumentation for the $r_{x,n}$ predicates. First, 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 $[\![\varphi_{r_{x,n}}]\!]_3^S([v \mapsto u])$ 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 for $r_{x,n}(v)$ and $c_n(v)$, are given in Table VIII and Table IX, 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. For example, the predicate-update formula in the `x->n = NULL` case in Table VIII, 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 on a directed cycle or is not reachable from `x`.

Let us now consider the predicate-update formulae for $r_{x,n}(v)$ that appear in Table VIII:

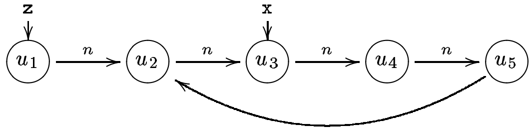


Fig. 11. For the statement `x->n = NULL`, the graph shown above illustrates the chief obstacle for updating reachability information: After the execution of `x->n = NULL`, the elements u_4 and u_5 are no longer reachable from `z`, whereas u_2 (and u_3) are still reachable from `z`. Note that beforehand the value of $r_{z,n}$ is the same for u_2, u_4, and u_5. For such individuals, Table VIII reevaluates $\varphi_{r_{z,n}}(v)$.

st	**cond.**	$\varphi^{st}_{r_{z,n}}(v)$
`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`	$z \equiv x$	$r_{t,n}(v) \wedge (c_n(v) \vee \neg t(v))$
	$z \not\equiv x$	$r_{z,n}(v)$
`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 that `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 VIII. The predicate-update formulae for the instrumentation predicate $r_{z,n}$, for programs that use the `List` data-type declaration from Fig. 1(a).

- 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 $r_{x,n}$ for all individuals for which $r_{t,n}$ holds, except for the individual u pointed to by `t`—unless u appears on a cycle, in which case $r_{x,n}(u)$ also holds.
- 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. 11, 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.
  - 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.

st	$\varphi_{c_n}^{st}(v)$
`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 that `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 IX. The predicate-update formulae for the instrumentation predicate c_n, for programs that use the `List` data-type declaration from Fig. 1(a).

Let us now examine the predicate-update formulae for $c_n(v)$ that appear in Table IX:

- The statements `x = NULL`, `x = t`, and `x = t->n` do not change the n-predicates, and thus have no effect 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 effect on cyclicity.
- If the node pointed to by `x` is reachable from `t`, 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.

6. ABSTRACT SEMANTICS

In this section, we formulate the abstract semantics for the shape-analysis framework. As an intermediate step, Sect. 6.1 first describes a simple abstract semantics based on the Reinterpretation Principle (Obs. 2.9). This version shows how the machinery developed thus far fits together, but serves mainly as a strawman: this first approach yields very imprecise information about programs that perform list traversals and destructive updates (such as `insert`), and this failing motivates the development of two new pieces of machinery (*focus* and *coerce*) to refine the strawman approach. The main ideas behind the more refined approach are sketched in Sect. 6.2, and formally defined in Sects. 6.3 and 6.4. Finally, Sect. 6.5 defines the more refined semantics, and also illustrates how it is capable of obtaining very precise shape-analysis information when applied to the analysis of `insert`.

6.1 A Strawman Shape-Analysis Algorithm

We now define a simple abstract semantics for the shape-analysis framework based on the Reinterpretation Principle (Obs. 2.9). The goal is to associate with each vertex v of control-flow graph G, a finite set of 3-valued structures *StructSet*$[v]$ that "describes" all of the 2-valued structures in *ConcStructSet*$[v]$ (and possibly more). The abstract semantics can be expressed as the least fixed point (in terms of set inclusion) of the following system of equations over the variables $\mathit{StructSet}[v]$:

$$\mathit{StructSet}[v] = \begin{cases} \{\langle\emptyset,\emptyset\rangle\} & \text{if } v = \mathit{start} \\ \left.\begin{array}{l} \displaystyle\bigcup_{\substack{w \to v \in E(G), \\ w \in As(G)}} \{t\_embed_c([\![st(w)]\!]_3(S)) \mid S \in \mathit{StructSet}[w]\} \\ \cup \displaystyle\bigcup_{\substack{w \to v \in E(G), \\ w \in Id(G)}} \{S \mid S \in \mathit{StructSet}[w]\} \\ \cup \displaystyle\bigcup_{w \to v \in Tb(G)} \{S \mid S \in \mathit{StructSet}[w] \text{ and } S \models_3 cond(w)\} \\ \cup \displaystyle\bigcup_{w \to v \in Fb(G)} \{S \mid S \in \mathit{StructSet}[w] \text{ and } S \models_3 \neg cond(w)\} \end{array}\right\} & \text{otherwise} \end{cases} \tag{20}$$

This equation system closely resembles equation system (4), but has the following differences:

—The notation $[\![st(w)]\!]_3$ denotes the abstract meaning function for statement w; it is identical to the operation $[\![st(w)]\!]$ defined in Defns. 3.3 and 3.5, except that the predicate-update formulae are evaluated in 3-valued logic.

—Instead of working with concrete 2-valued structures, equation system (20) operates on bounded structures and applies t_embed_c after every statement.

—When a formula $cond(w)$ evaluates to 1/2, equation system (20) conservatively propagates information along both the true and the false branches.

The shape-analysis algorithm takes the form of an iterative procedure that finds the least fixed point of equation system (20). The iteration starts from the initial assignment $\mathit{StructSet}[v] = \emptyset$ for each control-flow-graph vertex v. Because of the t_embed_c operation, it is possible to check efficiently if two 3-valued structures are isomorphic.

Termination and safety of the shape-analysis algorithm are argued in the standard manner [Cousot and Cousot 1977]. The algorithm terminates because, for any instantiation of the analysis framework, the set of bounded structures is finite, and hence of finite height. Termination is assured because equation system (20) is monotonic (with respect to set inclusion).

The heart of the safety argument involves showing that the abstract transformer that is applied along each edge of the control-flow graph is conservative with respect to the corresponding transformer of the concrete semantics:

THEOREM 6.1. [**Local Safety Theorem**]. *If vertex w is a condition, then for all $S \in$ 3-STRUCT$[\mathcal{P} \cup \{sm\}]$*

i. If $S^{\natural} \in \gamma(S)$ and $S^{\natural} \models cond(w)$, then $S \models_3 cond(w)$.

ii. If $S^{\natural} \in \gamma(S)$ and $S^{\natural} \models \neg cond(w)$, then $S \models_3 \neg cond(w)$.

If vertex w is a statement, then

iii. If $S^{\natural} \in \gamma(S)$, then $[\![st(w)]\!](S^{\natural}) \in \gamma(t\_embed_c([\![st(w)]\!]_3(S)))$.

Proof: By Defn. 4.8, $S^\natural \in \gamma(S)$ means that there is a mapping f such that $S^\natural \sqsubseteq^f S$. Thus, properties *i.* and *ii.* follow immediately from the Embedding Theorem.

If vertex w is a statement, then it follows from the Embedding Theorem and the definitions of $[\![st(w)]\!]$ and $[\![st(w)]\!]_3$ (Defns. 3.3 and 3.5) that, for any structure S,

—If $S^\natural \in \gamma(S)$, then $[\![st(w)]\!](S^\natural) \in \gamma([\![st(w)]\!]_3(S))$.

In particular, this means that there is a mapping f such that $[\![st(w)]\!](S^\natural) \sqsubseteq^f [\![st(w)]\!]_3(S)$. However, t_embed_c simply folds together and renames individuals from $[\![st(w)]\!]_3(S)$, so we know that the composed mapping $(t\_embed_c \circ f)$ embeds $[\![st(w)]\!](S^\natural)$ into $t\_embed_c([\![st(w)]\!]_3(S))$:

$$[\![st(w)]\!](S^\natural) \sqsubseteq^{(t\_embed_c \circ f)} t\_embed_c([\![st(w)]\!]_3(S)).$$

By the definition of γ (Defn. 4.8), this implies property *iii.*

THEOREM 6.2. **[Global Safety Theorem]**. *Let ConcStructSet*$[\cdot]$ *and StructSet*$[\cdot]$ *be the least-fixed-point solutions of equation systems (4) and (20), respectively. Then, for each vertex v of the control-flow graph,*

$$ConcStructSet[v] \subseteq \bigcup_{S \in StructSet[v]} \gamma(S).$$

Proof: Immediate from monotonicity and Theorem 6.1 (Local Safety), using Theorem T2 of [Cousot and Cousot 1977, p. 252].

Other variations on equation system (20) are possible. In particular, the function t_embed_c need not be applied after every statement; instead, it could be applied (i) at every merge point in the control-flow graph, or (ii) only in loops, e.g., at the target of each backedge in the control-flow graph. It is also possible to define a Hoare ordering on sets of structures that is induced by the embedding order:

DEFINITION 6.3. *For sets of structures* $XS_1, XS_1 \subseteq$ *3-STRUCT*$[\mathcal{P}]$, $XS_1 \sqsubseteq XS_2$ *if and only if* $\forall S_1 \in XS_1 : \exists S_2 \in XS_2 : S_1 \sqsubseteq S_2$.

In principle, this could be used to remove non-maximal structures during the course of the analysis; the shape-analysis algorithm would then be an iterative procedure to compute a least fixed point with respect to the Hoare order. However, this would require being able to test the property $S_1 \sqsubseteq S_2$ (i.e., whether there exists a function f that embeds S_1 into S_2).

Unfortunately, the use of the Reinterpretation Principle alone leads to shape-analysis algorithms that return very imprecise information. Fig. 12 shows the result of applying $[\![st_0]\!]_3$ to structure S_a—one of the 3-valued structures that arises in the analysis of `insert` just before `y` is advanced down the list by the statement st_0 : `y = y->n`. Similar to what was illustrated in Fig. 9, the structure S_b that results is not as precise as what the abstract domain of canonical abstractions is capable of representing: for instance, S_b does not contain a node that is definitely pointed to by `y`.

This imprecision leads to problems when a destructive update is performed. In particular, the first column in Table X shows what happens when the abstract transformers for the five statements that follow the search loop in `insert` are applied to S_b: Because $y(v)$ evaluates to $1/2$ for the summary node, we eventually

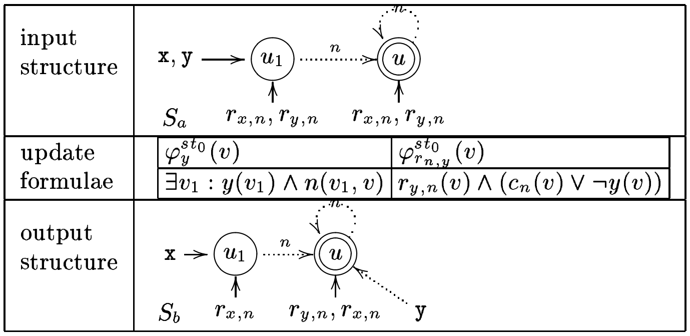


Fig. 12. An application of the strawman abstract transformer, $[\![st_0]\!]_3$, for statement st_0: `y = y->n` in `insert`.

reach the situation shown in the fifth row of structures, in which y, e, r_x, r_y, r_e, r_t, and is are all 1/2 for the summary node. As a result, under the strawman approach, the abstract transformer for `y->n = t` sets the value of c_n for the summary node to 1/2. Consequently, the strawman analysis fails to determine that the structure returned by `insert` is an acyclic list.

In contrast, the refined analysis described in the following subsections is able to determine that at the end of `insert` the following properties always hold: (i) `x` points to an acyclic list that has no shared elements, (ii) `y` points into the tail of the `x`-list, and (iii) the value of `e` and `y->n` are equal.

It is worthwhile to note that the precision problem becomes even more acute for shape-analysis algorithms that, like [Chase et al. 1990], do not explicitly track reachability properties. The reason is that, without reachability, S_b represents situations in which `y` points to an element that is not even part of the `x`-list.

6.2 An Overview of a More Precise Abstract Semantics

In formulating an improved approach, our goal is to retain an important property of the strawman approach, namely that the transformer for a program statement falls out automatically from the predicate-update formulae of the concrete semantics and the predicate-update formulae supplied for the instrumentation predicates. Thus, the main idea behind the more refined approach is to decompose the transformer for st into a composition of several functions, as depicted in Fig. 13 and explained below, each of which falls out automatically from the predicate-update formulae of the concrete semantics and the predicate-update formulae supplied for the instrumentation predicates:

(1) The operation *focus*, defined in Sect. 6.3, refines 3-valued structures so that the formulae that define the meaning of st evaluate to definite values. The *focus* operation thus brings these formulae "into focus".

(2) The simple abstract meaning function for statement st, $[\![st]\!]_3$, is then applied.

(3) The operation *coerce*, defined in Sect. 6.4, converts a 3-valued structure into a more precise 3-valued structure by removing certain kinds of inconsistencies.

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

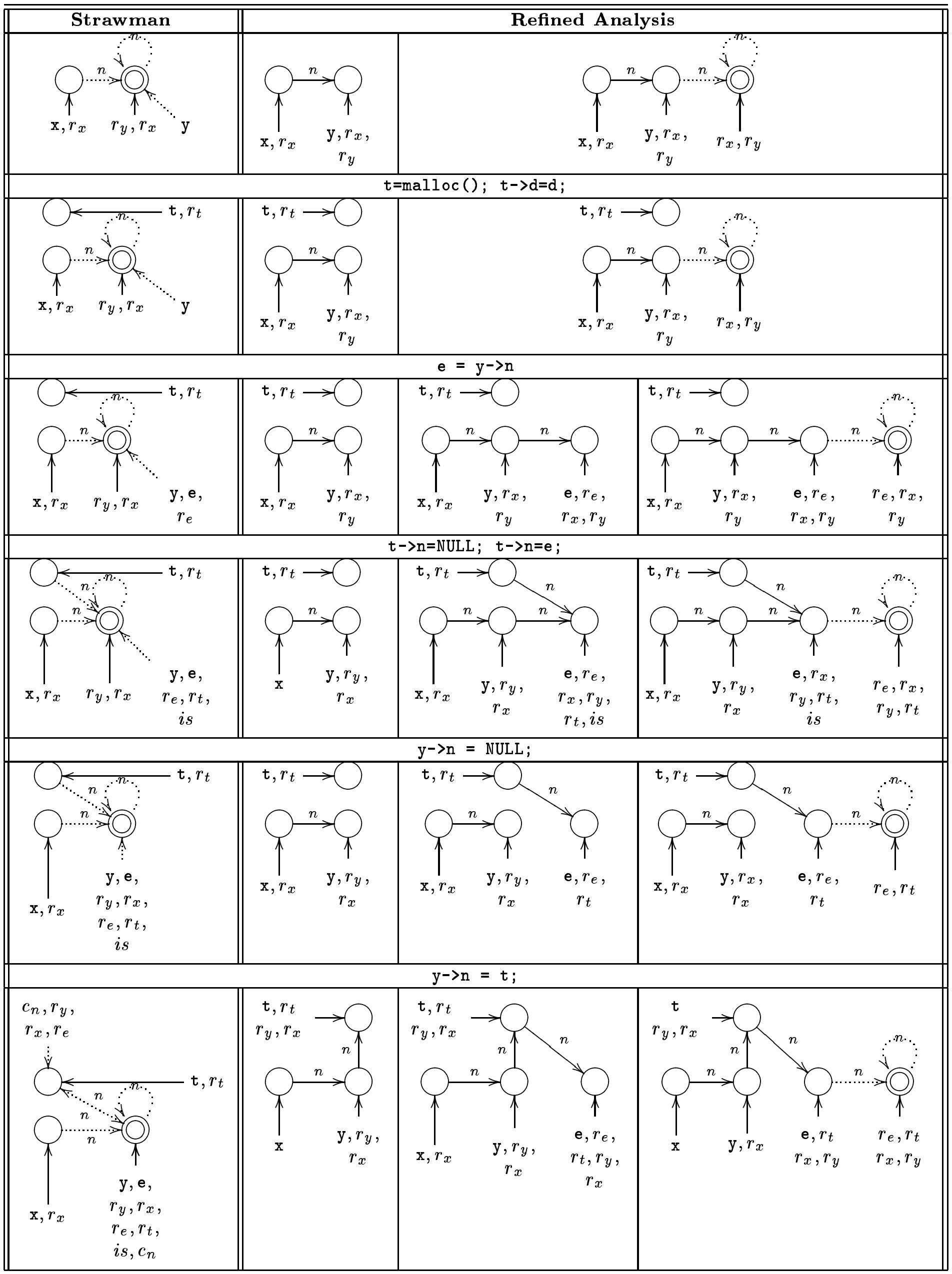


Table X. Selective applications of the abstract transformers using the Strawman and the Refined approaches, for the statements in `insert` that come after the search loop. (For brevity, r_z is used in place of $r_{z,n}$ for all variables `z`, and node names are not shown.)

$$\begin{tikzcd}
\{S_a\} \arrow[rrr, "focus"] \arrow[d, "\llbracket st_0 \rrbracket_3"'] & & & \{S_{a,f,0}, S_{a,f,1}, S_{a,f,2}\} \arrow[d, "\llbracket st_0 \rrbracket_3"] \\
\{S_b\} & \sqsupseteq & \{S_{b,1}, S_{b,2}\} & \{S_{a,o,0}, S_{a,o,1}, S_{a,o,2}\} \arrow[l, "coerce"']
\end{tikzcd}$$

Fig. 13. One-stage vs. multi-stage abstract semantics for statement st_0 : `y = y->n`. The *focus* and the *coerce* operations are introduced in Sects. 6.3 and 6.4, respectively. (This example will be discussed in further detail in Sects. 6.3 and 6.4.)

convert a set of 3-valued structures into a more precise set of 3-valued structures that describe the same set of stores. This property, together with the correctness of the structure transformer $\llbracket st \rrbracket_3$, guarantees that the overall multi-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.

It is also interesting to compare this approach to the best abstract transformer defined in [Cousot and Cousot 1979], which is obtained by applying the concrete transformer to every concrete store that the input 3-valued structure S represents (i.e., to all of the 2-valued structures in $\gamma(S)$). The best abstract transformer is conceptually simpler and potentially more precise, but cannot be directly computed (since $\gamma(S)$ is potentially infinite). In contrast, *focus* yields a finite set of 3-valued structures that represent the same concrete stores as S, and thus serves as a "partial concretization function." Since $\llbracket st \rrbracket_3$ is conservative by the Embedding Theorem, the overall result is guaranteed to be conservative. Our experience has been that having *focus* ensure that the "important" formulae have definite values is sufficient to ensure that the overall result is precise enough.

In contrast to *focus*, *coerce* does not depend on the particular statement *st*; it can be applied at any step, and may improve the precision of the analysis. In practice, it is often beneficial to also perform an application of *coerce* just after the application of *focus* and before $\llbracket st \rrbracket_3$, so that the order of operations becomes *focus*, *coerce*, $\llbracket st \rrbracket_3$, *coerce*—followed by t_embed_c.

6.3 Bringing Formulae Into Focus

In this section, 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 assignments. Unfortunately, in general *focus* may yield an infinite set of structures. Therefore, in Sect. 6.3.1, we give a declarative specification of the desired properties of the *focus* operation, and in Sect. 6.3.2, 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.

6.3.1 *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, $\llbracket st \rrbracket_3$, etc.),

$$\widehat{op}(XS) \stackrel{\text{def}}{=} \bigcup_{S \in XS} op(S). \tag{21}$$

DEFINITION 6.4. *Given a set of formulae F, a function op: $3\text{-}STRUCT[\mathcal{P}] \to 2^{3\text{-}STRUCT[\mathcal{P}]}$ is* **a focus operation for** *F if for every $S \in 3\text{-}STRUCT[\mathcal{P}]$, $op(S)$ satisfies the following requirements:*

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

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

In the above definition, Z maps the free variables of a formula $\varphi \in F$ to individuals in structures $S' \in op(S)$. In particular, when φ has one designated free variable v, Z maps v to an individual. As usual, when φ is a closed formula, the quantification over Z is superfluous.

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 structures that are maximal (in terms of the embedding order). However, in some cases the set of maximal structures is infinite, as well. This phenomenon is illustrated by the following example:

EXAMPLE 6.5. 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 $S_{acyclic}$ shown in Fig. 6 will lead to an infinite set of maximal structures (lists 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 Sect. 6.3.2 and illustrated in the following example.

EXAMPLE 6.6. For the statement st_0: `y = y->n` in procedure `insert`, we focus on the formula

$$\varphi_0(v) \stackrel{\text{def}}{=} \exists v_1 : y(v_1) \wedge n(v_1, v), \tag{22}$$

which corresponds to the R-value of st_0 (the heap cell pointed to by `y->n`). The upper part of Fig. 14 illustrates the application of $focus_{\{\varphi_0\}}(S_a)$, where S_a is the structure shown in Fig. 12 that occurs in `insert` just before the first application of statement st_0: `y = y->n`. This results in three structures: $S_{a,f,0}$, $S_{a,f,1}$, and $S_{a,f,2}$.

—In $S_{a,f,0}$, $[\![\varphi_0]\!]_3^{S_{a,f,0}}([v \mapsto u])$ 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. (As we will see later, this structure is actually inconsistent because of the values of the $r_{x,n}$ and $r_{y,n}$ instrumentation predicates, and will be eliminated from consideration by *coerce*.)

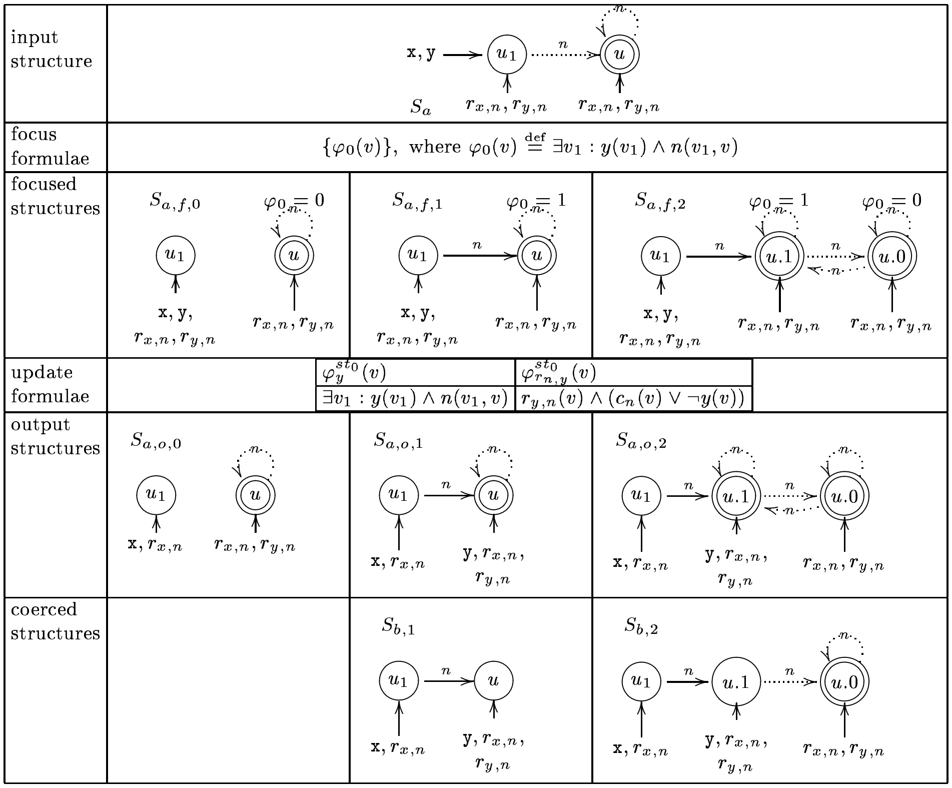


Fig. 14. The first application of the improved transformer for statement st_0: `y = y->n` in `insert`.

- In $S_{a,f,1}$, $[\![\varphi_0]\!]_3^{S_{a,f,1}}([v \mapsto u])$ 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 $S_{a,f,2}$, $[\![\varphi_0]\!]_3^{S_{a,f,2}}([v \mapsto u.0])$ equals 0 and $[\![\varphi_0]\!]_3^{S_{a,f,2}}([v \mapsto u.1])$ 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.

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

- $S_{a,f,0}$, in which $\iota^{S_{a,f,0}}(n)(u_1, u)$ is set to 0, and thus $[\![\varphi_0]\!]_3^{S_{a,f,0}}([v \mapsto u])$ equals 0.

—$S_{a,f,1}$, in which $\iota^{S_{a,f,1}}(n)(u_1, u)$ is set to 1, and thus $[\![\varphi_0]\!]_3^{S_{a,f,1}}([v \mapsto u])$ equals 1.

—$S_{a,f,2}$, in which u has been bifurcated into two different individuals, $u.0$ and $u.1$. In $S_{a,f,2}$, $\iota^{S_{a,f,2}}(n)(u_1, u.0)$ is set to 0, and thus $[\![\varphi_0]\!]_3^{S_{a,f,2}}([v \mapsto u.0])$ equals 0, whereas $\iota^{S_{a,f,2}}(n)(u_1, u.1)$ is set to 1, and thus $[\![\varphi_0]\!]_3^{S_{a,f,2}}([v \mapsto u.1])$ equals 1.

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

6.3.2 *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, 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. Focusing on L-values and R-values ensures that the application of the abstract transformer does not set to 1/2 the entries of core predicates that correspond to pointer variables and fields that are updated by the statement. This approach extends naturally to program conditions and to statements that manipulate multiple L-values and R-values.

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

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

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`	$\{x(v)\}$
`x != NULL`	$\{x(v)\}$
`x == t`	$\{x(v), t(v)\}$
`x != t`	$\{x(v), t(v)\}$
`UninterpretedCondition`	$\emptyset$

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

Fig. 15 contains an algorithm that implements *focus* for the type of formulae that arise in Table XI.[11] Here, 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. As shown in Table XI, in shape analysis there are two types of formulae that must be considered:

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

—$\varphi \equiv \exists v_1 : x(v_1) \wedge n(v_1, v)$, for $\mathtt{x} \in PVar$. In this case, FocusVarDeref$(S, \mathtt{x}, \mathtt{n})$ 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 6.7. Consider the application of FocusVarDeref$(S_a, \mathtt{y}, n)$ for structure S_a from Fig. 14. In this case, FocusVar$(S_a, \mathtt{y}) = \{S_a\}$. When S_a is selected from WorkSet, structures $S_{a,f,0}$, $S_{a,f,1}$, and $S_{a,f,2}$ 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. 15 is correct.

Lemma 6.8. *For $\varphi \equiv x(v)$, and for every structure $S \in$ 3-STRUCT$[\mathcal{P}]$,* FocusVar$(S, \mathtt{y})$ *is a focus operation for $\{\varphi\}$.*

Lemma 6.9. *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$(S, \mathtt{y}, \mathtt{n})$ *is a focus operation for $\{\varphi\}$.*

---

[11] An enhanced *focus* algorithm, which generalizes the methods describe here, is described in [Lev-Ami 2000]. This algorithm can be applied to an arbitrary formula, but may not always succeed.

```
function FocusVar($S_0$ : 3-STRUCT[$\mathcal{P}$], 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[$\mathcal{P}$], x : $PVar$, n : Selector) returns $2^{\text{3-STRUCT}[\mathcal{P}]}$
begin
  WorkSet := FocusVar($S_0$, 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 $m = \lambda u'. \begin{cases} u & \text{if } u' = u.0 \lor u' = u.1 \\ u' & \text{otherwise} \end{cases}$ 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. 15. An algorithm for *focus* for the two types of formulae that arise in Table XI.

It is not hard to see that both FocusVar and FocusVarDeref always return a finite (although not necessarily maximal) set of structures.

We use Focus to denote the operation that invokes FocusVar or FocusVarDeref, as appropriate.

6.4 Coercing into More Precise Structures

In this section, we define the operation *coerce*, which converts a 3-valued structure into a more precise 3-valued structure by removing certain kinds of inconsistencies.

EXAMPLE 6.10. After *focus*, the simple transformer $[\![st]\!]_3$ is applied to each of the structures produced. In the example discussed in Examples 6.6 and 6.7, $[\![st_0]\!]_3$ is applied to structures $S_{a,f,0}$, $S_{a,f,1}$, and $S_{a,f,2}$ to obtain structures $S_{a,o,0}$, $S_{a,o,1}$, and $S_{a,o,2}$, respectively (see Fig. 14).

However, this process 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. In particular, consider structure $S_{a,o,2}$. In this structure, the n field of $u.0$ can point to $u.1$, which suggests that y may be pointing to a heap-shared cell. However, this is incompatible with the fact that $\iota(is)(u.1) = 0$—i.e., $u.1$ cannot represent a heap-shared cell—and the fact that $\iota(n)(u_1, u.1) = 1$—i.e., it is known that $u.1$ definitely has an incoming n edge from a cell other than $u.0$.

Also, the structure $S_{a,o,0}$ describes an impossible situation: $\iota(r_{y,n})(u) = 1$ and yet u is not reachable—or even potentially reachable—from a heap cell that is pointed to by y.

In this section, we develop a systematic way to capture interdependences among the properties stored in 3-valued structures. The mechanism that we present removes indefinite values that violate certain consistency rules, thereby "sharpening" the structures that arise during shape analysis. This allows us to remedy the imprecision illustrated in Example 6.10. In particular, when the sharpening process is applied to structure $S_{a,o,2}$ from Fig. 14, the structure that results is $S_{b,2}$. In this case, the sharpening process discovers that (i) two of the n-edges with value 1/2 can be removed from $S_{a,o,2}$, and (ii) individual $u.1$ can only ever represent a single individual in each of the structures that $S_{a,o,2}$ represents, and hence $u.1$ should not be labeled as a summary node. These facts are not something that the mechanisms that have been described in earlier sections are capable of discovering. Also, the structure $S_{a,o,0}$ is discarded by the sharpening process.

The sharpening mechanism is crucial to the success of the improved shape-analysis framework because it allows a more accurate job of materialization to be performed than would otherwise be possible. For instance, note how the sharpened structure, $S_{b,2}$, clearly represents an unshared list of length 3 or more that is pointed to by x and whose second element is pointed to by y. In fact, in the abstract domain of canonical abstractions that is being used in our examples, $S_{b,2}$ is the most precise representation possible for the family of unshared lists of length 3 or more that are pointed to by x and whose second element is pointed to by y. Without the sharpening mechanism, instantiations of the framework would rarely be able to determine such things as "The data structure being manipulated by a certain list-manipulation program is actually a list."

This subsection is organized as follows: Sect. 6.4.1 discusses how structures should obey certain consistency rules. Sect. 6.4.2 discusses how we can obtain a system of "compatibility constraints" that formalize such consistency rules; the constraint system is obtained automatically from formulae that express certain global invariants on concrete stores. Compatibility constraints are used in Sect. 6.4.3 to define an operation, called *coerce*, that "coerces" a structure into a more precise structure. Finally, in Sect. 6.4.4, we give an algorithm for *coerce*.

6.4.1 *Compatibility Constraints.* We can, in many cases, sharpen some of the stored predicate values of 3-valued structures:

EXAMPLE 6.11. 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 Embedding Theorem (Theorem 4.9), $\iota^{S^\natural}(is)(u^\natural)$ 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 $\iota^{S'}(is)(u)$ 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 Defn. 4.6. This has operational significance: It is needlessly imprecise to work with structure S' in which $\iota^{S'}(is)(u)$ has the value 1/2; instead, we should discard S' and work with S. In general, the "stored predicate" 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. In $S_{a,o,0}$, for instance, $\varphi_{r_{y,n}}(u) = 0$, whereas $\iota^{S_{a,o,0}}(r_{y,n})(u)$ is 1; consequently, $S_{a,o,0}$ is a 3-valued structure that does not represent any concrete structures at all! This structure can therefore be eliminated from further consideration by the abstract-interpretation algorithm.

This reasoning applies to all instrumentation predicates, not just is and $r_{x,n}$, and to both of the definite values, 0 and 1.

The reasoning used in Example 6.11 can be summarized as the following principle:

OBSERVATION 6.12. [**The Sharpening Principle**]. *In any structure S, the value stored for $\iota^S(p)(u_1, \ldots, u_k)$ 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 $\iota^S(p)(u_1, \ldots, u_k)$ 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 connective, '$\triangleright$'.

DEFINITION 6.13. *A* **compatibility constraint** *is a term of the form $\varphi_1 \triangleright \varphi_2$, where φ_1 is an arbitrary 3-valued formula, and φ_2 is either an atomic formula or the negation of an atomic formula over distinct logical variables.*

We say that a 3-valued structure S and an assignment Z **satisfy** *$\varphi_1 \triangleright \varphi_2$, denoted by $S, Z \models \varphi_1 \triangleright \varphi_2$, if whenever Z is an assignment such that $[\![\varphi_1]\!]_3^S(Z) = 1$, we also have $[\![\varphi_2]\!]_3^S(Z) = 1$. (Note that if $[\![\varphi_1]\!]_3^S(Z)$ equals 0 or 1/2, S and Z satisfy $\varphi_1 \triangleright \varphi_2$, regardless of the value of $[\![\varphi_2]\!]_3^S(Z)$.)*

We say that S satisfies $\varphi_1 \triangleright \varphi_2$, denoted by $S \models \varphi_1 \triangleright \varphi_2$, if for every Z we have $S, Z \models \varphi_1 \triangleright \varphi_2$. If Σ is a finite set of compatibility constraints, we write $S \models \Sigma$ if

S satisfies every constraint in Σ.

The compatibility constraint that captures the reasoning used in Example 6.11 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 in order to satisfy the constraint. The compatibility constraint used to capture the similar case of sharpening $\iota(is)(u)$ from 1/2 to 0 is $\neg\varphi_{is}(v) \rhd \neg is(v)$.

Sect. 6.4.4 presents a constraint-satisfaction algorithm that repeatedly searches for assignments Z such that $S, Z \not\models \varphi_1 \rhd \varphi_2$ (i.e., $[\![\varphi_1]\!]_3^S(Z) = 1$, but $[\![\varphi_2]\!]_3^S(Z) \neq 1$). This algorithm is used to improve the precision of shape analysis by (i) sharpening the values of predicates stored in S (when the constraint violation is repairable), and (ii) eliminating S from further consideration when the constraint violation is irreparable.

6.4.2 *From Formulae to Constraints.* Compatibility constraints provide a way to express certain properties that are a consequence of the tight-embedding process, but that would not be expressible with formulae alone. 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., $S, Z \not\models \varphi_1 \rhd \varphi_2$).

In general, compatibility 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 4.4), whereas $\rhd$ is non-monotonic in its right-hand-side argument. For instance, the constraint $\mathbf{1} \rhd p$ is satisfied in the structure $S = \langle \emptyset, [p \mapsto 1] \rangle$; however, it is not satisfied in $S' = \langle \emptyset, [p \mapsto 1/2] \rangle \sqsupseteq S$.

Thus, in 3-valued logic, compatibility constraints are in some sense "better" than formulae. Fortunately, compatibility constraints can be generated automatically from formulae that express certain global invariants on concrete stores. We call such formulae *compatibility formulae.* There are two sources of compatibility formulae:

- The formulae that define the instrumentation predicates.
- Additional formulae ("hygiene conditions") that formalize the properties of stores that are compatible with the semantics of C (i.e., with our encoding of C stores as 2-valued logical structures).

In the remainder of the paper, 2-CSTRUCT$[\mathcal{P}, F]$ denotes the set of *compatible 2-valued structures* that satisfy a given set of compatibility formulae F.[12]

The following definition supplies a way to convert formulae into constraints:

Definition 6.14. *Let φ be a closed formula, and (where applicable below) let a be an atomic formula such that (i) a contains no repetitions of logical variables,*

---

[12] We also use a weaker notion of when a predicate-update formula for p maintains the correct instrumentation for statement st; in particular, in Defn. 5.2, the occurrence of 2-STRUCT$[\mathcal{P}]$ is replaced by 2-CSTRUCT$[\mathcal{P}, F]$.

The notion of concretization (Defn. 4.8) is adjusted in the same manner.

and (ii) $a \not\equiv sm(v)$. Then the **constraint generated from** φ*, denoted by* $r(\varphi)$*, is defined as follows:*

$$r(\varphi) = \varphi_1 \rhd a \qquad \text{if } \varphi \equiv \forall v_1, \dots v_k : (\varphi_1 \Rightarrow a) \tag{23}$$

$$r(\varphi) = \varphi_1 \rhd \neg a \qquad \text{if } \varphi \equiv \forall v_1, \dots v_k : (\varphi_1 \Rightarrow \neg a) \tag{24}$$

$$r(\varphi) = \neg\varphi \rhd \mathbf{0} \qquad \text{otherwise} \tag{25}$$

For a set of formulae F, we define $\widehat{r}(F)$ to be the set of constraints generated from the formulae in F (i.e., $\{r(\varphi) \mid \varphi \in F\}$).

The intuition behind (23) and (24) 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 $v_1, \dots v_k$, 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 6.15 below). Rule (25) is included to enable an arbitrary formula to be converted to a constraint.

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

LEMMA 6.15. *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: See Appendix C.

EXAMPLE 6.16. It is worthwhile to point out 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

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

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

$$\iota^\natural = [sm \mapsto [u_1 \mapsto 0, u_2 \mapsto 0], p_1 \mapsto [u_1 \mapsto 1, u_2 \mapsto 0], p_2 \mapsto [u_1 \mapsto 0, u_2 \mapsto 1]]$$

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, $S = \langle \{u_{1,2}\}, \iota \rangle)$, where

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

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

Compatibility Constraints from Instrumentation Predicates. Our first source of compatibility formulae is the set of formulae that define the instrumentation predicates. For every instrumentation predicate $p \in \mathcal{I}$ defined by a formula $\varphi_p(v_1, \dots, v_k)$, we generate a compatibility formula of the following form:

$$\forall v_1, \dots, v_k : \varphi_p(v_1, \dots, v_k) \Leftrightarrow p(v_1, \dots, v_k) \tag{26}$$

So that we can apply Defn. 6.14, this is then broken into two implications:

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

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

For instance, for the instrumentation predicate is, we use formula (13) for φ_{is} to generate compatibility formulae (27) and (28), which lead to compatibility constraints (33) and (34) (see Table XII).

$$
\begin{array}{rcll}
\forall v : (\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) & \Rightarrow & is(v) & (27) \\
\forall v : \neg(\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) & \Rightarrow & \neg is(v) & (28) \\
\forall v : n^+(v, v) & \Rightarrow & c_n(v) & (29) \\
\forall v : \neg n^+(v, v) & \Rightarrow & \neg c_n(v) & (30) \\
\text{for each } \mathtt{x} \in \mathit{PVar}, \forall v : x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v) & \Rightarrow & r_{x,n}(v) & (31) \\
\text{for each } \mathtt{x} \in \mathit{PVar}, \forall v : \neg(x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v)) & \Rightarrow & \neg r_{x,n}(v) & (32) \\
\hline
(\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) & \rhd & is(v) & (33) \\
\neg(\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) & \rhd & \neg is(v) & (34) \\
n^+(v, v) & \rhd & c_n(v) & (35) \\
\neg n^+(v, v) & \rhd & \neg c_n(v) & (36) \\
\text{for each } \mathtt{x} \in \mathit{PVar} : x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v) & \rhd & r_{x,n}(v) & (37) \\
\text{for each } \mathtt{x} \in \mathit{PVar} : \neg(x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v)) & \rhd & \neg r_{x,n}(v) & (38)
\end{array}
$$

Table XII. The set of formulae listed above the line are the compatibility formulae generated for the instrumentation predicates is, c_n, and $r_{x,n}$. The corresponding compatibility constraints are listed below the line.

$$
\begin{array}{rcll}
 & & \neg\exists v : sm(v) & (39) \\
\text{for each } \mathtt{x} \in \mathit{PVar}, \forall v_1, v_2 : x(v_1) \wedge x(v_2) & \Rightarrow & v_1 = v_2 & (40) \\
\forall v_1, v_2 : (\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2)) & \Rightarrow & v_1 = v_2 & (41) \\
\hline
(\exists v : sm(v)) & \rhd & \mathbf{0} & (42) \\
\text{for each } \mathtt{x} \in \mathit{PVar}, x(v_1) \wedge x(v_2) & \rhd & v_1 = v_2 & (43) \\
(\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2)) & \rhd & v_1 = v_2 & (44)
\end{array}
$$

Table XIII. The set of formulae listed above the line, denoted by $F_{\mathtt{List}}$, are compatibility formulae for structures that represent a store of a C program that operates on values of the type `List` defined in Fig. 1(a). The corresponding compatibility constraints $\widehat{r}(F_{\mathtt{List}})$ are listed below the line.

Compatibility Constraints from Hygiene Conditions. Our second source of compatibility formulae stems from the fact that 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.

EXAMPLE 6.17. The set of formulae $F_{\mathtt{List}}$, listed above the line in Table XIII, is a set of compatibility formulae that must be satisfied for a structure to represent a store of a C program that operates on values of the type `List` defined in Fig. 1(a). Formula (39) captures the condition that concrete stores never contain any summary nodes. Formula (40) captures the fact that every program variable points to at most one list element. Formula (41) 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. The corresponding compatibility constraints generated according to Defn. 6.14 are listed below the line.

Compatibility Constraints from "Extended Horn Clauses". The constraint-generation rules defined in Defn. 6.14 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 compatibility formulae written as implications (cf. Tables XII and XIII), the set of constraints generated depends on the form in which the compatibility formulae are written. In particular, not all of the many equivalent forms possible for a given compatibility formula 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 prevent an instantiation of the shape-analysis framework from having a suitable compatibility constraint at its disposal that would otherwise allow it to sharpen or discard a structure that arises during the analysis—and hence can lead to a shape-analysis algorithm that is more conservative than we would like. However, when compatibility formulae are written as "extended Horn clauses" (see Defn. 6.18 below), the way around this difficulty is to augment the constraint-generation process to generate constraints for some of the logical consequences of each compatibility formula. The process of "generating some of the logical consequences for extended Horn clauses" is formalized as follows:

Definition 6.18. *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(φ), to be the following set of formulae:*

$$closure(\varphi) \stackrel{\text{def}}{=} \left\{ \forall \ldots, \exists v_1, v_2, \ldots, v_n : \bigwedge_{i=1, i \neq j}^{m} \varphi_i^{1-B_i} \Rightarrow \varphi_j^{B_j} \;\middle|\; \begin{array}{l} 1 \leq j \leq m, \\ v_k \in freeVars(\varphi), \\ v_k \notin freeVars(\varphi_j) \end{array} \right\} \quad (45)$$

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 6.19. The set of formulae listed in Table XIV are the compatibility formulae $\widehat{closure}(F_{\texttt{List}})$ generated via Defn. 6.18 when the two implication formulae in $F_{\texttt{List}}$, (40) and (41), are expressed as the following extended Horn clauses (i.e., by rewriting the implications as disjunctions, and then applying De Morgan's laws):

$$\text{for each } \texttt{x} \in PVar, \forall v_1, v_2 : \neg x(v_1) \vee \neg x(v_2) \vee v_1 = v_2 \quad (50)$$

$$\forall v_1, v_2, v_3 : \neg n(v_3, v_1) \vee \neg n(v_3, v_2) \vee v_1 = v_2 \quad (51)$$

From (50) and (51), Defn. 6.18 generates the final six compatibility formulae shown

$$\neg\exists v : sm(v) \tag{39}$$

$$\text{for each } \mathtt{x} \in PVar, \forall v_1, v_2 : x(v_1) \wedge x(v_2) \Rightarrow v_1 = v_2 \tag{40}$$

$$\text{for each } \mathtt{x} \in PVar, \forall v_2 : (\exists v_1 : x(v_1) \wedge v_1 \neq v_2) \Rightarrow \neg x(v_2) \tag{46}$$

$$\text{for each } \mathtt{x} \in PVar, \forall v_1 : (\exists v_2 : x(v_2) \wedge v_1 \neq v_2) \Rightarrow \neg x(v_1) \tag{47}$$

$$\forall v_1, v_2 : (\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2)) \Rightarrow v_1 = v_2 \tag{41}$$

$$\forall v_2, v_3 : (\exists v_1 : n(v_3, v_1) \wedge v_1 \neq v_2) \Rightarrow \neg n(v_3, v_2) \tag{48}$$

$$\forall v_1, v_3 : (\exists v_2 : n(v_3, v_2) \wedge v_1 \neq v_2) \Rightarrow \neg n(v_3, v_1) \tag{49}$$

Table XIV. The compatibility formulae $\widehat{closure}(F_{\mathtt{List}})$ generated via Defn. 6.18 when the two implication formulae in $F_{\mathtt{List}}$, (40) and (41), are expressed as the extended Horn clauses (50) and (51), respectively. (Note that the systematic application of Defn. 6.18 leads, in this case, to two pairs of formulae that differ only in the names of their bound variables: (46)/(47) and (48)/(49).)

in Table XIV. By Defn. 6.14 these yield the following compatibility constraints:

$$\text{for each } \mathtt{x} \in PVar, x(v_1) \wedge x(v_2) \rhd v_1 = v_2 \tag{43}$$

$$\text{for each } \mathtt{x} \in PVar, (\exists v_1 : x(v_1) \wedge v_1 \neq v_2) \rhd \neg x(v_2) \tag{52}$$

$$\text{for each } \mathtt{x} \in PVar, (\exists v_2 : x(v_2) \wedge v_1 \neq v_2) \rhd \neg x(v_1) \tag{53}$$

$$(\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2)) \rhd v_1 = v_2 \tag{44}$$

$$(\exists v_1 : n(v_3, v_1) \wedge v_1 \neq v_2) \rhd \neg n(v_3, v_2) \tag{54}$$

$$(\exists v_2 : n(v_3, v_2) \wedge v_1 \neq v_2) \rhd \neg n(v_3, v_1) \tag{55}$$

Similarly, after (27) is rewritten as the following extended Horn clause

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

we obtain the following compatibility constraints:

$$(\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \rhd is(v) \tag{33}$$

$$(\exists v_1 : n(v_1, v) \wedge v_1 \neq v_2 \wedge \neg is(v)) \rhd \neg n(v_2, v) \tag{56}$$

$$(\exists v_2 : n(v_2, v) \wedge v_1 \neq v_2 \wedge \neg is(v)) \rhd \neg n(v_1, v) \tag{57}$$

$$(\exists v : n(v_1, v) \wedge n(v_2, v) \wedge \neg is(v)) \rhd v_1 = v_2 \tag{58}$$

As we will see in Sect. 6.4.4, the use of constraints—and, in particular, the ones created from formulae generated by *closure*—plays a crucial role in the shape-analysis framework. In particular, constraint (56) (or the equivalent constraint (57)) allows a more accurate job of materialization to be performed than would otherwise be possible: When $is(u)$ is 0 and one incoming n edge to u is 1, to satisfy constraint (56) 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 6.10 and 6.26). This allows edges to be removed (safely) that a more naive materialization process would retain (cf. structures $S_{a,o,2}$ and $S_{b,2}$ in Fig. 14), and permits the improved shape-analysis algorithm to generate more precise structures for `insert` than the ones generated by the simple shape-analysis algorithm described in Sects. 2 and 6.1.

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

DEFINITION 6.20. **(Compatible 3-Valued Structures)**. *Given a set of compatibility formulae 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 6.21. *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) $S^\natural \in \gamma(S')$*
Proof: Let $S^\natural \in \gamma(S)$, then by Defn. 4.8 (as modified by footnote 12), $S^\natural \models F$ 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 6.15, S' satisfies the necessary requirements.

In Sect. 6.4.4, we give an algorithm that constructs from S and $\widehat{r}(F)$ a maximal S' meeting the conditions of Lemma 6.21 (without investigating the possibly infinite set of actual concrete structures $S^\natural \in \gamma(S)$).

6.4.3 *The Coerce Operation.* We are now ready to show how the *coerce* operation works.

EXAMPLE 6.22. Consider structure $S_{a,o,2}$ from Fig. 14. This structure violates constraint (56) for the assignment $[v \mapsto u.1, v_2 \mapsto u.0]$ when the variable v_1 of the existential quantifier is bound to u_1: because $\iota(n)(u_1, u.1) = 1$, $u_1 \neq u.0$, and $\iota(is)(u.1) = 0$, but $\iota(n)(u.0, u.1) = 1/2$, constraint (56) 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 6.23. *The operation*

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

is defined as follows: $coerce_{\widehat{r}(F)}(S) \stackrel{\text{def}}{=}$ 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.

(We will simply write *coerce* when $\widehat{r}(F)$ is clear from the context.)
It is a fact that the maximal such structure S' is unique (if it exists), which follows from the observation that compatible structures with the same universe of individuals are closed under the following join operation:

DEFINITION 6.24. *For every pair of structures $S_1, S_2 \in 3\text{-}CSTRUCT[\mathcal{P}, \widehat{r}(F)]$ 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:*

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

LEMMA 6.25. *For every pair of structures $S_1, S_2 \in 3\text{-}CSTRUCT[\mathcal{P}, \widehat{r}(F)]$ such that $U^{S_1} = U^{S_2} = U$, the structure $S_1 \sqcup S_2$ is also in $3\text{-}CSTRUCT[\mathcal{P}, \widehat{r}(F)]$.*
Proof: See Appendix C.

Because *coerce* can result in at most one structure, its definition does not involve 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 6.26. The application of *coerce* to the structures $S_{a,o,0}, S_{a,o,1}$, and $S_{a,o,2}$ yields $S_{b,1}$ and $S_{b,2}$, as shown in the bottom block of Fig. 14.

- The structure $S_{a,o,0}$ is discarded because there exists no structure that can be embedded into it that satisfies constraint (38).
- The structure $S_{b,1}$ was obtained from $S_{a,o,1}$ by removing incompatibilities as follows:
  1. Consider the assignment $[v \mapsto u, v_1 \mapsto u_1, v_2 \mapsto u]$. Because $\iota(n)(u_1, u) = 1$, $u_1 \neq u$, and $\iota(is)(u) = 0$, constraint (56) implies that $\iota(n)(u, u)$ must equal 0. Thus, in $S_{b,1}$ the (indefinite) `n` edge from u to u has been removed.
  2. Consider the assignment $[v_1 \mapsto u, v_2 \mapsto u]$. Because $\iota(y)(u) = 1$, constraint (43) implies that $[\![v_1 = v_2]\!]_3^{S_{b,1}}([v_1 \mapsto u, v_2 \mapsto u])$ must equal 1. By Defn. 4.2, this means that $\iota^{S_{b,1}}(sm)(u)$ must equal 0. Thus, in $S_{b,1}$ u is no longer a summary node.
- The structure $S_{b,2}$ was obtained from $S_{a,o,2}$ by removing incompatibilities as follows:
  1. Consider the assignment $[v \mapsto u.1, v_1 \mapsto u_1, v_2 \mapsto u.0]$. Because $\iota(n)(u_1, u.1) = 1$, $u_1 \neq u.0$, and $\iota(is)(u.1) = 0$, constraint (56) implies that $\iota^{S_{b,2}}(n)(u.0, u.1)$ must equal 0. Thus, in $S_{b,2}$ 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 $\iota(n)(u_1, u.1) = 1$, $u_1 \neq u.1$, and $\iota(is)(u.1) = 0$, constraint (56) implies that $\iota^{S_{b,2}}(n)(u.1, u.1)$ must equal 0. Thus, in $S_{b,2}$ 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 $\iota(y)(u.1) = 1$, constraint (43) implies that $[\![v_1 = v_2]\!]_3^{S_{b,2}}([v_1 \mapsto u.1, v_2 \mapsto u.1])$ must equal 1. By Defn. 4.2, this means that $\iota^{S_{b,2}}(sm)(u.1)$ must equal 0. Thus, in $S_{b,2}$ $u.1$ is no longer a summary node.

There are important differences between the structures $S_{b,1}$ and $S_{b,2}$ that result from applying the refined abstract transformer for statement st_0 : `y = y->n`, compared with the structure S_b that results from applying the strawman abstract transformer (see Fig. 12). For instance, `y` points to a summary node in S_b, whereas `y` does not point to a summary node in either $S_{b,1}$ or $S_{b,2}$; as noted earlier, in the abstract domain of canonical abstractions that is being used in our examples, $S_{b,2}$ is the most precise representation possible for the family of unshared lists of length 3 or more that are pointed to by `x` and whose second element is pointed to by `y`.

6.4.4 *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 Defn. 6.13. It is convenient to partition these constraints into the following types:

$$\varphi(v_1, v_2, \ldots, v_k) \ \triangleright \ \mathbf{0} \tag{59}$$

$$\varphi(v_1, v_2, \ldots, v_k) \ \triangleright \ (v_1 = v_2)^b \tag{60}$$

$$\varphi(v_1, v_2, \ldots, v_k) \ \triangleright \ p^b(v_1, v_2, \ldots, v_k) \tag{61}$$

where $p \neq sm$, and the superscript notation used is the same as in Defn. 6.18: $\varphi^1 \equiv \varphi$ and $\varphi^0 \equiv \neg\varphi$. We say that constraints in the forms (59), (60), and (61) are *Type I*, *Type II*, and *Type III* constraints, respectively.

The Coerce algorithm is shown in Fig. 16. 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 $\bot$, 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 the literal **0**.
- 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 when the right-hand side is an equality that evaluates to 1/2 can be fixed. This type of violation occurs 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.

EXAMPLE 6.27. When the Coerce algorithm is applied to $S_{a,o,0}$, the Type III constraint (38) for program variable y is irreparably violated, and $\bot$ is returned.

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 6.28. *For every $S \in$ 3-STRUCT[$\mathcal{P}$], $coerce_{\widehat{r}(F)}(S) = \mathrm{Coerce}(S, \widehat{r}(F))$.* Proof: See Appendix C.

6.5 The Shape-Analysis Algorithm

In this section, we define the more refined abstract semantics, which includes applications of *focus* and *coerce*. (The actual algorithm uses Focus and Coerce.) The main idea is to refine equation system (20) by performing *focus* at the beginning of each abstract transformer that is applied along an edge of the control-flow graph,

```
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 $\varphi_2$
      case $\varphi_2 \equiv \mathbf{0}$ /* 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. 16. An iterative algorithm for solving 3-valued constraints.

and *coerce* at the end. Formally, this is defined as follows:

$$
\mathit{StructSet}[v] = \begin{cases} \{\langle\emptyset,\emptyset\rangle\} & \text{if } v = \mathit{start} \\ \left.\begin{array}{l} \displaystyle\bigcup_{\substack{w \to v \in E(G),\\ w \in As(G)}} \widehat{t\_embed}_c(\widehat{coerce}(\llbracket \widehat{st(w)} \rrbracket_3(\widehat{focus}_{F(w)}(\mathit{StructSet}[w])))) \\ \displaystyle\cup \bigcup_{\substack{w \to v \in E(G),\\ w \in Id(G)}} \{S \mid S \in \mathit{StructSet}[w]\} \\ \displaystyle\cup \bigcup_{w \to v \in Tb(G)} \left\{ t\_embed_c(S) \;\middle|\; \begin{array}{r} S \in \widehat{coerce}(\widehat{focus}_{F(w)}(\mathit{StructSet}[w])) \\ \text{and } S \models_3 \mathit{cond}(w) \end{array} \right\} \\ \displaystyle\cup \bigcup_{w \to v \in Fb(G)} \left\{ t\_embed_c(S) \;\middle|\; \begin{array}{r} S \in \widehat{coerce}(\widehat{focus}_{F(w)}(\mathit{StructSet}[w])) \\ \text{and } S \models_3 \neg\mathit{cond}(w) \end{array} \right\} \end{array}\right\} & \text{otherwise} \end{cases} \tag{62}
$$

Here $F(w)$ is the set of focus formulae for w (see Table XI).

As with the strawman semantics, the safety argument involves showing that the abstract transformer that is applied along each edge of the control-flow graph is conservative with respect to the corresponding transformer of the concrete semantics. Because *focus* and *coerce* are defined as semantic reductions, their presence in equation system (62) merely serves to increase the precision of the final answer. As before, the safety of the uses of $\llbracket\cdot\rrbracket_3$ and $\models_3$ follow from the Embedding Theorem. Formally, we show the following local safety theorem:

THEOREM 6.29. [**Local Safety Theorem**]. *If vertex w is a condition, then for all $S \in$ 3-STRUCT$[\mathcal{P} \cup \{sm\}]$*

i. If $S^\natural \in \gamma(S)$ and $S^\natural \models \mathit{cond}(w)$, then there exists $S' \in \widehat{coerce}(\widehat{focus}_{F(w)}(S))$ such that $S' \models_3 \mathit{cond}(w)$ and $S^\natural \in \gamma(t\_embed_c(S'))$.

ii. If $S^\natural \in \gamma(S)$ *and* $S^\natural \models \neg cond(w)$*, then there exists* $S' \in \widehat{coerce}(focus_{F(w)}(S))$ *such that* $S' \models_3 \neg cond(w)$ *and* $S^\natural \in \gamma(t\_embed_c(S'))$.

If vertex w *is a statement, then*

iii. If $S^\natural \in \gamma(S)$*, then* $[\![st(w)]\!](S^\natural) \in \widehat{\gamma}(t\_\widehat{embed}_c(\widehat{coerce}([\![\widehat{st(w)}]\!]_3(focus_{F(w)}(S)))))$.

Proof: See Appendix C.

The global safety property is argued in the same way as in the strawman semantics.

Example 6.30. Table XV shows the 3-valued structures that occur before and after applications of the abstract transformer for the statement `y = y->n` during the abstract interpretation of `insert`. (Because we are analyzing a single procedure, we allow an arbitrary set of 3-valued structures to hold at the entry of the procedure, as opposed to equation system (62), which assumes a single, empty initial structure. The global safety theorem still holds as long as all of the initial concrete structures are represented by the 3-valued structures that are provided for the entry point.)

The material in Table X that appears under the heading "Refined Analysis" shows the application of the abstract transformers for the five statements that follow the search loop in `insert` to $S_{b,1}$ and $S_{b,2}$. For space reasons, we do not show the abstract execution of these statements on the other structures shown in Table XV; however, the analysis is able to determine that at the end of `insert` the following properties always hold: (i) `x` points to an acyclic list that has no shared elements, (ii) `y` points into the tail of the `x`–list, and (iii) the value of `e` and `y->n` are equal. The identification of the latter condition is rather remarkable: the analysis is capable of showing that `e` and `y->n` are must-aliases at the end of `insert`.

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]. 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,

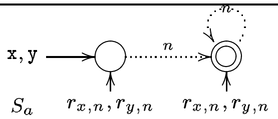
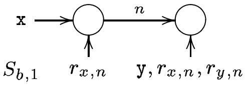
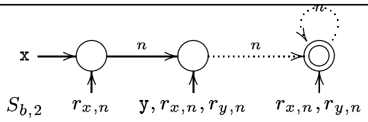
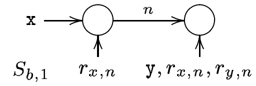
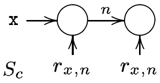
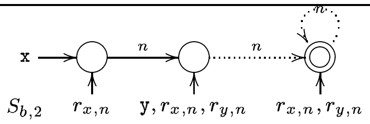
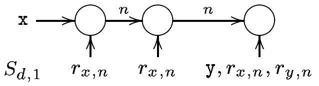
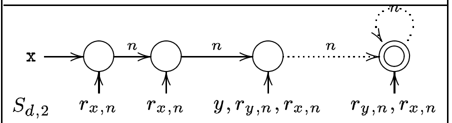
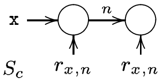
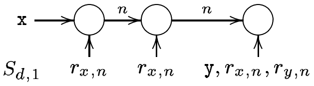
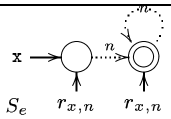
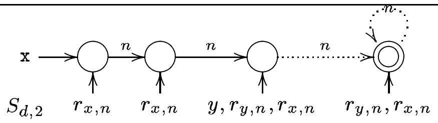
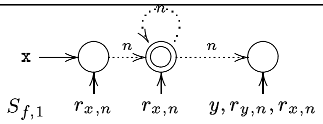
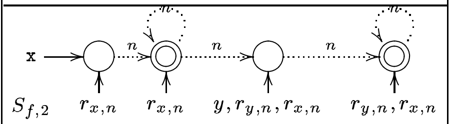
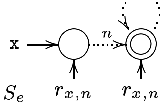
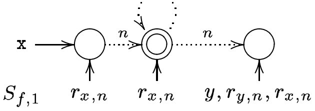
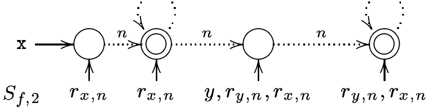
Iter.	Structure Before	Structures After
1		 
2		
3		 
4		$= S_c$
5		
6		 
7		$= S_e$
8		$= S_e$
9		$= S_{f,1}$ $= S_{f,2}$

Table XV. The structures that occur before and after successive applications of the abstract transformer for the statement `y = y->n` during the abstract interpretation of `insert`. (For brevity, node names are not shown.)

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.

Of course, a trade-off is involved: with our approach it is necessary to define the instrumentation predicates that are appropriate for a given analysis. It is also usually necessary to provide predicate-update formulae that specify how the values of instrumentation predicates are affected by the execution of each kind of statement in the programming language, and to prove that these formulae maintain the correct instrumentation (in the sense of Defn. 5.2 and footnote 12). It is open to debate whether these are more or less burdensome tasks than the ones that one faces with more standard approaches to abstract interpretation.

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 Analysis

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 Sect. 4.3, 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 Tables V, VI, and Appendix A).

Throughout this paper, we have focused on precision and ignored efficiency. Some of the above-cited algorithms are more efficient than instantiations of the framework presented in this paper because they keep only a single abstract structure at each program point. However, this issue has been addressed in the TVLA system, which implements the 3-valued logic framework (see Sect. 7.4.1). 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* (and *coerce*).

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]. In [Elgaard et al. 2000], this method was extended to handle programs that use tree data structures. The method is complete for loop-free code, but for loops and recursive functions it relies on Hoare-style invariants. In contrast, the shape-analysis framework described in the present paper can handle some programs that manipulate shared structures (as well as some circular structures, such as doubly linked lists). Because a set of 3-valued structures produced by shape analysis can also be viewed as a representation of a data-structure invariant, our approach can be thought of as providing a way to synthesize loop invariants. For certain instantiations of the framework, the invariants obtained for programs that manipulate linked lists and doubly linked lists are rather precise.

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 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 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.

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 Embedding and Canonical Abstraction

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 found in the literature is the notion of embedding discussed in [Bell and Machover 1977, p. 165]. For Bell and Machover, 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: 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*.

In Sect. 4, there were two steps involved in defining a suitable family of fixed-size abstract structures:

- Sect. 4.2 introduced "truth-blurring" onto mappings, for which 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. In particular, a *tight embedding* is one that minimizes the information lost in mapping concrete individuals to abstract individuals.
- Sect. 4.3 introduced *canonical abstractions*, which were defined as the tight embeddings induced by t_embed_c. The use of t_embed_c ensures that the result of embedding is a *bounded structure*, and hence of *a priori* finite size.

Canonical abstraction is related to the notion of *predicate abstraction* introduced in [Graf and Saidi 1997], and used subsequently by others [Das et al. 1999; Clarke et al. 2000]. However, canonical abstraction yields 3-valued predicates, whereas predicate abstraction yields 2-valued predicates. Moreover, the use of 3-valued predicates provides some additional flexibility; in particular, it permits canonical abstraction to mesh with the more general notion of embedding. This ability was important for the material presented in Sect. 6, which discussed a number of improvements to the abstract semantics; the methods developed in Sect. 6 take advantage of the ability to work, at times, with structures that are not "bounded" in the sense of Defn. 4.11.

7.4 Follow-On Work Using 3-Valued Logic

The work presented in the paper was originally motivated by the problem of shape analysis—how to determine shape invariants of programs that perform destructive updating on dynamically allocated storage. The paper explains how the various ingredients that are part of the analysis framework can be used to specify (intraprocedural) shape-analysis algorithms, as well as how to fine-tune the precision of such algorithms.

It is important to understand, however, that the material presented in the paper actually has a much broader range of applicability to program-analysis problems in general: as has been shown in work that has been carried out subsequent to the original presentation of the approach in [Sagiv et al. 1999], the use of 3-valued logic is not restricted just to shape analysis. In fact, the machinery of 3-valued logic—embedding, tight embedding, bounded structures, canonical abstraction, *focus*, and *coerce*—provides a framework in which a wide variety of program-analysis problems can be addressed. Below, we summarize the results that have been obtained to date in several pieces of follow-on work.

7.4.1 *TVLA.* The approach presented in this paper has been implemented by T. Lev-Ami in a system called *TVLA*, for *Three-Valued Logic Analysis* (see [Lev-Ami 2000; Lev-Ami and Sagiv 2000]). TVLA provides a language in which the user can specify (i) an operational semantics (via predicates and predicate-update formulae), (ii) a control-flow graph for a program, and (iii) a set of 3-valued structures that describe the program's input. From this specification, TVLA builds the corresponding equation system, and finds its least fixed point (cf. Eqn. (20)).

TVLA was used to test out the ideas described in the present paper. The experience gained from this effort led to a number of improvements to, and extensions of, the methods that have been described here. Some of the enhancements that TVLA incorporates include

—The ability to declare that certain binary predicates specify *functional properties.*

—The ability to specify that structures should be stored only at nodes of the control-flow graph that are targets of backedges.

—An enhanced version of Coerce that exploits dependences among the set of constraints to speed up the constraint-satisfaction process.

—An enhanced *focus* algorithm that generalizes the methods of Sect. 6.3 to handle focusing on arbitrary formula.[13] In addition, this version of *focus* also takes advantage of the properties of predicates that are specified to be functions.

—The ability to specify criteria for merging together structures associated with a program point. This feature is motivated by the idea that when the number of structures that arise at a given program point is too large, it may be better to create a smaller number of structures that represent at least the same set of 2-valued structures.
In particular, nullary predicates (i.e., predicates of 0-arity) are used to specify which structures are to be merged together. For example, for linked lists, the "`x`-is-not-null" predicate, defined by the formula $nn[x]() = \exists v : x(v)$, discriminates between structures in which `x` actually points to a list element, and structures in which it does not. By using $nn[x]()$ as the criterion for whether to merge structures, the structures in which `x` is `NULL` are kept separate from those in which `x` points to an allocated memory cell.

Further details about these features can be found in [Lev-Ami 2000].

7.4.2 *Verification of Sorting Implementations.* In [Lev-Ami et al. 2000], 3-valued logic is applied to the problems of

—Automatically proving the partial correctness of correct programs.

—Discovering, locating, and diagnosing bugs in incorrect programs.

An algorithm is presented that analyzes sorting programs that manipulate linked lists. The main idea is to refine the list abstraction that was introduced in Sect. 2.6 by adding two more predicates:

---

[13] The enhanced *focus* algorithm may not always succeed.

- Core predicate $dle(v_1, v_2)$ records the fact that the value in the data-field of element v_1 is less than or equal to the value in the data-field of element v_2.
- Instrumentation predicate $inOrder(v)$ holds for memory cells that are connected in increasing order in a linked list. (To reduce the cost of the analysis, this instrumentation predicate was not used as an abstraction predicate.)

The TVLA implementation of the algorithm was found to be sufficiently precise to discover that (correct versions) of bubble-sort and insertion-sort procedures do, in fact, produce correctly sorted lists as outputs, and that the invariant "is-sorted" is maintained by list-manipulation operations, such as element-insertion, element-deletion, and even destructive list reversal and merging of two sorted lists. When the algorithm was run on erroneous versions of bubble-sort and insertion-sort procedures, it was capable of discovering and, in some cases, locating and diagnosing the error.

7.4.3 *Interprocedural Analysis.* In [Rinetskey and Sagiv 2001], the problem of interprocedural shape analysis for programs with recursive procedures is addressed. The main idea is to expose the run-time stack as an explicit "data structure" of the operational semantics; that is, activation records are individuals, and suitable core predicates are introduced to capture how activation records are linked together to form a stack. Instrumentation predicates are used to record information about the calling context and the "invisible" copies of variables in pending activation records on the stack. The resulting algorithm is expensive, but quite precise. For example, it can show the absence of memory leaks in a recursive implementation of a list-reversal procedure that, in turn, uses a recursive version of a list-append procedure.

7.4.4 *Analyzing Mobile Ambients.* In [Nielson et al. 2000], 3-valued logic is applied to the problem of analyzing mobile ambients [Cardelli and Gordon 1998]. The challenge here is that the number of 3-valued structures arising in the analysis is quite large. In this case, the ability to specify a criterion for merging together structures was crucial to the success of the analysis. Using this feature, the implementation of the analysis in TVLA was able to verify non-trivial properties of a routing program, including mutual exclusion.

7.4.5 *Checking Multithreaded Systems.* In [Yahav 2001], it is shown how to apply 3-valued logic to the problem of checking properties of multithreaded systems. In particular, [Yahav 2001] addresses the problem of state-space exploration for languages, such as Java, that allow (i) dynamic creation and destruction of an unbounded number of threads, (ii) dynamic allocation and freeing of an unbounded number of storage cells from the heap, and (iii) destructive updating of structure fields. This combination of features creates considerable difficulties for any method that tries to check program properties.

In the present paper, the problem of program analysis is expressed as a problem of annotating a control-flow graph with sets of 3-valued structures; in contrast, the analysis algorithm given in [Yahav 2001] is one that builds and explores a 3-valued transition system on-the-fly.

In [Yahav 2001], problems (ii) and (iii) are handled essentially via the techniques developed in the present paper; problem (iii) is addressed by reducing it to problem (ii): threads are modeled by individuals, which are abstracted using canonical names—in this case, the collection of unary thread properties that hold for a given thread. The use of this naming scheme automatically discovers commonalities in the state space, but without relying on explicitly supplied symmetry properties, as in, for example, [Emerson and Sistla 1993; Clarke and Jha 1995].

Unary core predicates are used to represent the program counter of each thread object; *focus* implements the interleaving of threads. The analysis described in [Yahav 2001] is capable of proving the absence of deadlock in a dining-philosophers program that permits there to be an unbounded number of philosophers.

In [Yahav et al. 2001], this approach is extended to provide a method for verifying LTL properties of multithreaded systems.

8. SOME FINAL OBSERVATIONS

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

8.1 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.[14] 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_c), 3-valued logical structures cannot grow to be of unbounded size.

The conventional approach to verifying programs that use data structures built using pointers is to characterize the structures in terms of invariants that describe their shape at stable points, i.e., outside of the procedures that may be applied to them [Hoare 1975]. Because data-structure invariants are usually temporarily violated within such procedures, it is challenging to prove that invariants are reestablished at the end of these procedures. As mentioned in Sect. 7.4.2, the analy-

---

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

sis approach developed in this paper has also been applied to a program-verification problem [Lev-Ami et al. 2000]. From the perspective of someone interested in analyzing or verifying a program via our approach, the need to adopt a local, element-wise view of a data structure gives the approach a markedly different flavor from the conventional approach to program verification. In particular, the notion of an instrumentation predicate can be contrasted with that of an invariant:

- A data-structure invariant states a global property of the instances of a data structure that holds on entry to and exit from the operations that can be performed on the data structure.
- An instrumentation predicate captures a local property that can be used to distinguish among some of a data structure's components.

As noted earlier, a set of 3-valued structures produced by the shape-analysis algorithm can also be viewed as a representation of a data-structure invariant; consequently, our approach can be thought of as providing a way to synthesize global invariants from local properties.

8.2 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.

Acknowledgments

We are grateful to T. Lev-Ami for his implementation of TVLA, and for his suggestions about improving the precision of the algorithms presented in the paper. We are also grateful for the helpful comments of A. Avron, T. Ball, M. Benedikt, N. Dor, M. Fahndrich,

J. Field, M. Gitik, K. Kunen, V. Lifschitz, F. Nielson, H.R. Nielson, M. O'Donnell, A. Rabinovich, N. Rinetskey, K. Sieber, and E. Yahav. The suggestions from the anonymous referees about how to improve the organization of the paper are also greatly appreciated. We thank K.H. Rose for the XY LaTeX package.

REFERENCES

AIKEN, A. 1996. Position statement. Prepared for the Programming Languages Working Group of the ACM Workshop on Strategic Directions in Computing Research, (Cambridge, MA, June 14-15, 1996).

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.

CARDELLI, L. AND GORDON, A. 1998. Mobile ambients. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS)*, M. Nivat, Ed. LNCS, vol. 1378. Springer-Verlag, 140–155.

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.

CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2000. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV '00).*

CLARKE, E. AND JHA, S. 1995. Symmetry and induction in model checking. In *Computer Science Today: Recent Trends and Developments*, J. V. Leeuwen, Ed. Lecture Notes in Computer Science, vol. 1000. Springer-Verlag.

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. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.* ACM Press, New York, NY, 238–252.

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.

DAS, S., DILL, D., AND PARK, S. 1999. Experience with predicate abstraction. In *Computer Aided Verification (CAV '99).* Springer-Verlag. Trento, Italy.

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

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

ELGAARD, J., MØLLER, A., AND SCHWARTZBACH, M. 2000. Compile-time debugging of c programs working on trees. In *Proc. of the 2000 European Symp. On Programming.* 119–134.

EMERSON, E. AND SISTLA, A. 1993. Symmetry and model checking. In *Proc. of The Fifth Workshop on Computer-Aided Verificaton (CAV '93)*, C. Courcoubetis, Ed.

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

GRAF, S. AND SAIDI, H. 1997. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV '97).*

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.

JENSEN, J., JØRGENSEN, 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.

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, Tel-Aviv, Israel.

LEV-AMI, T., REPS, T., SAGIV, M., AND WILHELM, R. 2000. Putting static analysis to work for verification: A case study. In *ISSTA 2000: Proc. of the Int. Symp. on Software Testing and Analysis.*

LEV-AMI, T. AND SAGIV, M. 2000. TVLA: A system for implementing static analyses. In *Static Analysis Symp.* 280–301.

LIFSCHITZ, V. 1998. Personal communication.

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. 2000. A Kleene analysis of mobile ambients. In *Proc. of ESOP 2000*, G. Smolka, Ed. LNCS, vol. 1782. Springer, 305–319.

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.

RINETSKEY, N. AND SAGIV, M. 2001. Interprocedural shape analysis for recursive programs. In *Proc. of CC 2001*, R. Wilhelm, Ed. LNCS, vol. 2027. Springer, 133–149.

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.* ACM Press, New York, NY, 105–118.

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.

YAHAV, E. 2001. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Symp. on Princ. of Prog. Lang.* ACM Press, New York, NY, 27–40.

YAHAV, E., REPS, T., AND SAGIV, M. 2001. LTL model checking for systems with unbounded number of dynamically created threads and objects. Tech. Rep. TR-1424, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. Mar.

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

(a)

```
/* 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;
}
```

(b)

Fig. 17. (a) Declaration of a doubly linked-list data type in C. (b) A program that splices an element with a data value `d` into a doubly linked list just 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`.

A. HANDLING DOUBLY LINKED LISTS

We now briefly sketch the treatment of doubly linked lists. A C declaration of a doubly-linked-list element is given in Fig. 17(a). 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 VI as equations (17) and (18):

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

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

The predicate-update formulae for $c_{f.b}$ are given in Table XVI. (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) $r_{z,f}(v)$, which holds for elements v that are reachable from `z` via 0 or more applications of the field-selector `f`, and (ii) $r_{z,b}(v)$, 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 VIII and Table IX (with n replaced by f and b). (One way in which Table VIII 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 overly 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 `insert` (see Table XV). The shape-analysis algorithm works in a similar fashion when a pointer is advanced along a doubly linked-list. Therefore, in this section, we consider the operation `splice`, shown in Fig. 17(b), which splices an element with a data value `d` into a doubly linked list just 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`.)

Table XVII illustrates the abstract interpretation of `splice` under the following

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`	$c_{f.b}(v) \vee x(v)$
`x->b = NULL`	$c_{f.b}(v) \wedge \neg\exists v_1 : x(v_1) \wedge b(v_1, v)$
`x->f = t` (assuming that `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 that `x->b == NULL`)	$c_{f.b}(v) \vee (t(v) \wedge \exists v_1 : f(v, v_1) \wedge x(v_1))$
`x = malloc()`	$c_{f.b}(v) \vee new(v)$

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

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 Table XVII, 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 by the last row of Table XVI, the value of $c_{f.b}$ for the newly allocated element 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.

—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 because the value of `e->f` is already `NULL`. The second stage changes the value of $c_{f.b}$ to 0 for the element pointed to by `e`, and changes the values of $r_{e,f}$ to 1 for the elements transitively pointed to by `t`.

—The assignment `t->b = e` is performed in two stages: (i) `t->b = NULL` and then (ii) `t->b = e`, assuming that `t->b == NULL`.
In the first stage, the assignment `t->b = NULL` changes the value of $c_{f.b}$ to 0 for the element pointed to by `p`. Also, the elements reachable in the backward direction from `t` are no longer reachable from `t`.
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`. Also, the nodes reachable in the backward direction from `e` are now reachable from `t`.

—The assignment `p->f = e` is performed in two stages: (i) `p->f = NULL` and then (ii) `p->f = e`, assuming that `p->f == NULL`.
In the first stage, the assignment `p->f = NULL` 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.

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

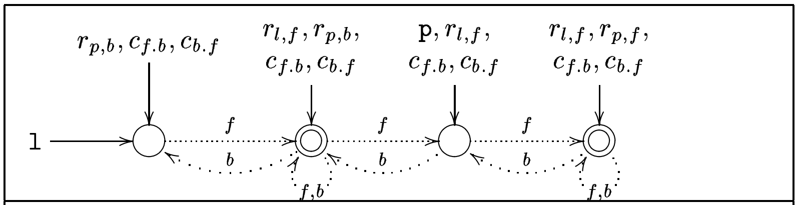
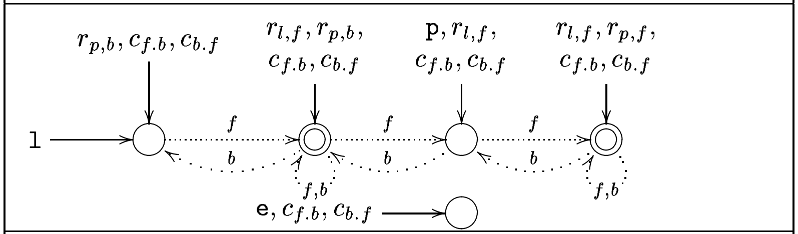
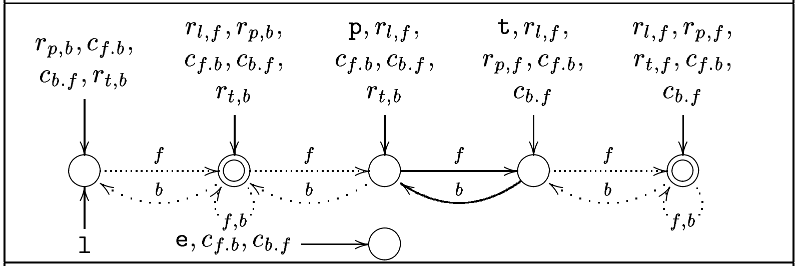
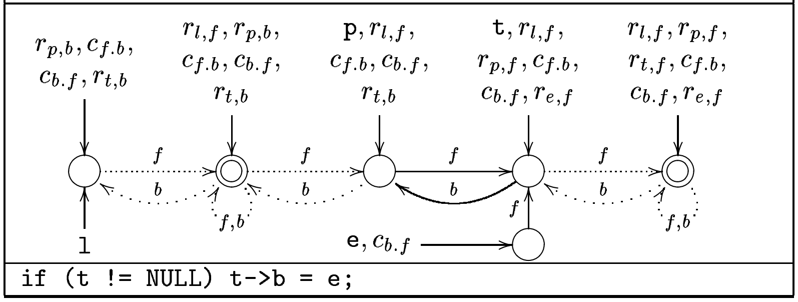
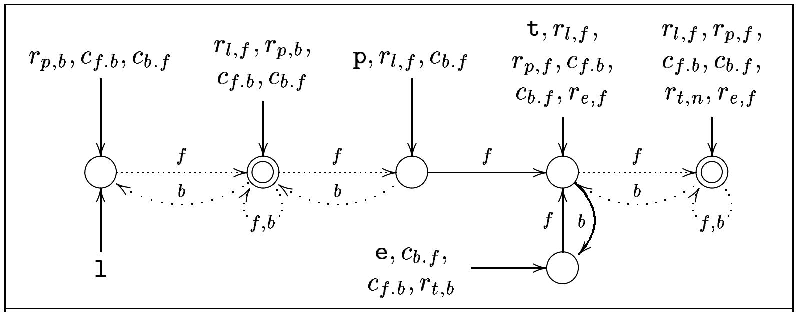
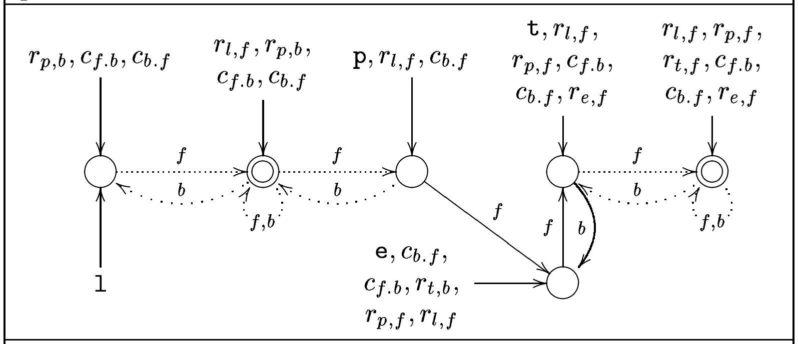

`e = (DList)malloc(sizeof(struct DListNode));`

`e->data = v; t = p->f;`

`e->f = t;`

`if (t != NULL) t->b = e;`

Table XVII: The abstract interpretation of the `splice` procedure applied to a doubly linked list whose head is pointed to by `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, node names are not shown, and $r_{z,f}(v)$ and $r_{z,b}(v)$ are omitted when v is directly pointed to by variable `z`.



`p->f = e;`



`e->b = p;`

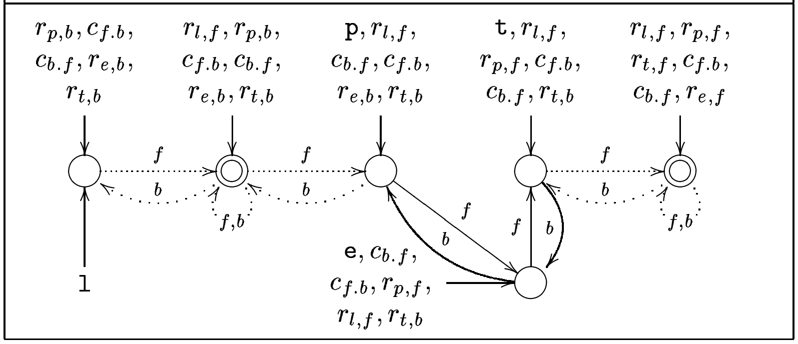


Table XVII: The abstract interpretation of the `splice` procedure applied to a doubly linked list whose head is pointed to by `l`. Variable `p` points to some element in the list beyond the second element, and the tail of `p` is assumed to be non-`NULL`. For brevity, node names are not shown, and $r_{z,f}(v)$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

B. PROOF OF THE EMBEDDING THEOREM

Theorem 4.9 *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 De Morgan's 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

$$\begin{aligned}
&[\![p(v_1, v_2, \ldots, v_k)]\!]_3^S(Z) \\
&\quad = \iota^S(p)(u_1, u_2, \ldots, u_k) && \text{(Defn. 4.2)} \\
&\quad \sqsubseteq \iota^{S'}(p)(f(u_1), f(u_2), \ldots, f(u_k)) && \text{(Defn. 4.5)} \\
&\quad = [\![p(v_1, v_2, \ldots, v_k)]\!]_3^{S'}(f \circ Z) && \text{(Defn. 4.2)}
\end{aligned}$$

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

$$\begin{aligned}
&[\![l]\!]_3^S(Z) \\
&\quad = l && \text{(Defn. 4.2)} \\
&\quad \sqsubseteq l && \text{(Defn. 4.1)} \\
&\quad = [\![l]\!]_3^{S'}(f \circ Z) && \text{(Defn. 4.2)}
\end{aligned}$$

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 Defn. 4.2, (i) $f(Z(v_1)) = f(Z(v_2))$ and (ii) $\iota^{S'}(sm)(f(Z(v_1))) = 0$. Therefore, by Defn. 4.5, $Z(v_1) = Z(v_2)$ and $\iota^S(sm)(Z(v_1)) = 0$ both hold. Hence, by Defn. 4.2, $[\![v_1 = v_2]\!]_3^S(Z) = 1$. Finally, suppose that $[\![v_1 = v_2]\!]_3^{S'}(f \circ Z) = 0$ holds. In this case, by Defn. 4.2, $f(Z(v_1)) \neq f(Z(v_2))$. Therefore, $Z(v_1) \neq Z(v_2)$, and by Defn. 4.2 $[\![v_1 = v_2]\!]_3^S(Z) = 0$.
Induction step: Suppose that 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 must consider four cases, according to the outermost operator of φ:

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 Defn. 4.2, $[\![\varphi_1 \wedge \varphi_2]\!]_3^S(Z) = 0$.
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 Defn. 4.2, $[\![\varphi_1 \wedge \varphi_2]\!]_3^S(Z) = 1$.

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 Defn. 4.2, $[\![\neg\varphi_1]\!]_3^S(Z) = 0$.
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 Defn. 4.2, $[\![\neg\varphi_1]\!]_3^S(Z) = 1$.

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 Defn. 4.2, $[\![\exists v_1 : \varphi_1]\!]_3^S(Z) = 0$.

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 Defn. 4.2, $[\![\exists v_1 : \varphi_1]\!]_3^S(Z) = 1$.

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 Defn. 4.2, 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 Defn. 4.2, $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^S(Z) = 1$.

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 Defn. 4.2 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 Defn. 4.2, $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_3^{S'}(f \circ Z) \neq 0$, which is a contradiction.

C. OTHER PROOFS

C.1 Properties of the Generated 3-Valued Constraints

Lemma 6.15 *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, …, 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, …, 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 that contains no repetitions of logical variables, $a \not\equiv sm(v)$, and $b \in \{0, 1\}$. Let Z be an assignment for v_1, v_2, …, v_k drawn from U^S. If $[\![\varphi]\!]_3^S(Z) \neq 1$, then by definition $S, Z \models \varphi \triangleright a^b$. Therefore, assume that $\llbracket\varphi\rrbracket_3^S(Z) = 1$ and let us show that $\llbracket a^b\rrbracket_3^S(Z) = 1$. Note that for every assignment $Z^\natural$ such that $f \circ Z^\natural = Z$, $\llbracket\varphi\rrbracket_3^S(Z) = 1$ implies, by the Embedding Theorem, that $\llbracket\varphi\rrbracket_3^{S^\natural}(Z^\natural) = 1$. Therefore, because $S^\natural \models \varphi'$, we have

$$\llbracket a^b\rrbracket_3^{S^\natural}(Z^\natural) = 1. \tag{63}$$

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 $l \leq k$, $p \in \mathcal{P} - \{sm\}$. Note that by Defn. 6.13, $i \neq j \Rightarrow v_i \neq v_j$. We have:

$$\begin{aligned}
&\llbracket p(v_1, v_2, \ldots, v_l)\rrbracket_3^S(Z) \\
&\quad = \iota^S(p)(Z(v_1), Z(v_2), \ldots, Z(v_l)) && \text{(Defn. 4.2)} \\
&\quad = \bigsqcup_{f(u_i^\natural)=Z(u_i)} \iota^{S^\natural}(p)(u_1^\natural, u_2^\natural, \ldots, u_l^\natural) && \text{(Defn. 4.6)} \\
&\quad = \bigsqcup_{f \circ Z^\natural=Z} \iota^{S^\natural}(p)(Z^\natural(v_1), Z^\natural(v_2), \ldots, Z^\natural(v_l)) && (\text{since } i \neq j \Rightarrow v_i \neq v_j) \\
&\quad = \bigsqcup_{f \circ Z^\natural=Z} \llbracket p(v_1, v_2, \ldots, v_l)\rrbracket_3^{S^\natural}(Z^\natural) && \text{(Defn. 4.2)} \\
&\quad = 1 && \text{(Eqn. (63))}
\end{aligned}$$

Notice that we use the fact that $p \not\equiv sm$ because the step from the third line to the fourth line may not hold for sm (cf. Defn. 4.6).

Case 2: $b = 0$ and $a \equiv p(v_1, v_2, \ldots, v_l)$, where $l \leq k$, $p \in \mathcal{P} - \{sm\}$. Again, by Defn. 6.13, $i \neq j \Rightarrow v_i \neq v_j$. We have:

$$\begin{aligned}
&\llbracket \neg p(v_1, v_2, \ldots, v_l)\rrbracket_3^S(Z) \\
&\quad = 1 - \iota^S(p)(Z(v_1), Z(v_2), \ldots, Z(v_l)) && \text{(Defn. 4.2)} \\
&\quad = 1 - \bigsqcup_{f(u_i^\natural)=Z(v_i)} \iota^{S^\natural}(p)(u_1^\natural, u_2^\natural, \ldots, u_l^\natural) && \text{(Defn. 4.6)} \\
&\quad = 1 - \bigsqcup_{f \circ Z^\natural=Z} \iota^{S^\natural}(p)(Z^\natural(v_1), Z^\natural(v_2), \ldots, Z^\natural(v_l)) && (\text{since } i \neq j \Rightarrow v_i \neq v_j) \\
&\quad = 1 - \bigsqcup_{f \circ Z^\natural=Z} \llbracket p(v_1, v_2, \ldots, v_l)\rrbracket_3^{S^\natural}(Z^\natural) && \text{(Defn. 4.2)} \\
&\quad = \bigsqcup_{f \circ Z^\natural=Z} \llbracket \neg p(v_1, v_2, \ldots, v_l)\rrbracket_3^{S^\natural}(Z^\natural) && \text{(Defn. 4.2)} \\
&\quad = 1 && \text{(Eqn. (63))}
\end{aligned}$$

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 (63). Now assume that $\iota(sm)(Z(v_1)) = 1/2$; thus, by Defn. 4.6 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 (63).

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 (63).

Lemma 6.25 *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 4.4), $[\![\varphi_1]\!]_3^{S_1}(Z) = 1$ 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.

C.2 Correctness of the Coerce Algorithm

The correctness of algorithm Coerce stems from the following two lemmas:

Lemma C.1. *For every $S \in$ 3-STRUCT$[\mathcal{P}]$ and structure S' before each iteration of the loop in* Coerce$(S, \widehat{r}(F))$, *the following conditions hold: (i) $S' \sqsubseteq S$; (ii) if coerce$_{\widehat{r}(F)}(S) \neq \bot$, then coerce$_{\widehat{r}(F)}(S) \sqsubseteq S'$.*

Proof: By induction on the number of iterations.

Basis: When the number of iterations is zero, the claim holds because (i) $S' = S$ and thus $S' \sqsubseteq S$, and (ii) if *coerce*$(S) \neq \bot$ then *coerce*$(S) \sqsubseteq S = S'$.

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 the loop in Coerce, and let $\varphi_1 \rhd \varphi_2$ and Z be the constraint and the assignment selected for the i^{th} iteration of the loop. We will show that the induction hypothesis still holds after the i^{th} iteration.

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

Part II. Let us show that (ii) holds. Assume that $S'' =$ *coerce*$(S) \neq \bot$. By part (ii) of the induction hypothesis, $S'' \sqsubseteq S'$. Because $[\![\varphi_1]\!]_3^{S'}(Z) = 1$, by Lemma 4.4 we have $[\![\varphi_1]\!]_3^{S''}(Z) = 1$. Hence, because $S'' \models \widehat{r}(F)$, it must be that $[\![\varphi_2]\!]_3^{S''}(Z) = 1$.

The proof splits into the following cases:

Case 1: $\varphi_2 \equiv v_1 = v_2$ and Coerce lowers $\iota^{S'}(sm)(Z(v_1))$ from 1/2 to 0. Because $[\![v_1 = v_2]\!]_3^{S''}(Z) = 1$, by Defn. 4.2 we know that $Z(v_1) = Z(v_2)$ and $\iota^{S''}(sm)(Z(v_1)) = 0$. Therefore, $S'' \sqsubseteq S'$ holds after the i^{th} iteration.

Case 2: $\varphi_2 \equiv p^b(v_1, v_2, \ldots, v_k)$ and Coerce lowers $\iota^{S'}(p)(Z(v_1), Z(v_2), \ldots, Z(v_k))$ from 1/2 to b. Because $[\![p^b(v_1, v_2, \ldots, v_k)]\!]_3^{S''}(Z) = 1$, by Defn. 4.2 we know that $\iota^{S''}(p)(Z(v_1), Z(v_2), \ldots, Z(v_k)) = b$. Therefore, $S'' \sqsubseteq S'$ holds after the i^{th} iteration.

Lemma C.2. *If* Coerce *returns $\bot$, then coerce$_{\widehat{r}(F)}(S) = \bot$.*

Proof: Let us assume that Coerce returns $\bot$, and yet $S'' =$ *coerce*$(S) \neq \bot$; we will show that this assumption leads to a contradiction.

Let S' be the structure at the beginning of the iteration of the loop in Coerce on which $\bot$ is returned, and let $\varphi_1 \rhd \varphi_2$ and Z be the constraint and the assignment selected for that iteration. By Lemma C.1(ii), $S'' \sqsubseteq S'$. Because $[\![\varphi_1]\!]_3^{S'}(Z) = 1$, by Lemma 4.4 we have $[\![\varphi_1]\!]_3^{S''}(Z) = 1$. Hence, because $S'' \models \widehat{r}(F)$, it must be that $[\![\varphi_2]\!]_3^{S''}(Z) = 1$. The proof splits into the following cases, according to what causes Coerce to return $\bot$:

Case 1: Coerce returns $\bot$ when a Type I constraint is violated. Immediate.

Case 2: Coerce returns $\bot$ when a Type II constraint is violated. There are two subcases to consider.

Case 2.1: $\varphi_2 \equiv v_1 = v_2$. Because $[\![v_1 = v_2]\!]_3^{S'}(Z) \neq 1$ and the constraint is irreparably violated, it must be the case that $Z(v_1) \neq Z(v_2)$. Therefore, by Defn. 4.2, $[\![v_1 = v_2]\!]_3^{S''}(Z) \neq 1$ — a contradiction.

Case 2.2: $\varphi_2 \equiv \neg(v_1 = v_2)$. Because $[\![\neg(v_1 = v_2)]\!]_3^{S'}(Z) \neq 1$, we conclude from Defn. 4.2 that $Z(v_1) = Z(v_2)$. Therefore, by Defn. 4.2, $[\![\neg(v_1 = v_2)]\!]_3^{S''}(Z) \neq 1$ — a contradiction.

Case 3: Coerce returns $\bot$ when a Type III constraint is violated. This happens when $\iota^{S'}(p)(Z(v_1), Z(v_2), \ldots, Z(v_k))$ has the definite value $1-b$. By Lemma C.1(ii), $S'' \sqsubseteq S'$, and therefore, by Lemma 4.4, $\iota^{S''}(p)(Z(v_1), Z(v_2), \ldots, Z(v_k))$ must also have the value $1 - b$. Therefore, by Defn. 4.2, $[\![p^b(v_1, v_2, \ldots, v_k)]\!]_3^{S''}(Z) = 0$ — a contradiction.

Theorem 6.28 *For every $S \in$ 3-STRUCT$[\mathcal{P}]$, $coerce_{\widehat{r}(F)}(S) = \text{Coerce}(S, \widehat{r}(F))$.*

Proof: Let T be the return value of Coerce$(S, \widehat{r}(F))$. There are two cases to consider, according to the value of T:

—Suppose $T = \bot$. By Lemma C.2, $coerce_{\widehat{r}(F)}(S) = \bot = T$.

—If $T \neq \bot$, then by Lemma C.1(i), $T \sqsubseteq S$. By the definition of the Coerce algorithm, $T \models \widehat{r}(F)$, and therefore, by Defn. 6.23, $coerce_{\widehat{r}(F)}(S) \neq \bot$. Consequently, by Lemma C.1(ii), $coerce_{\widehat{r}(F)}(S) \sqsubseteq T$. By Defn. 6.23, $coerce_{\widehat{r}(F)}(S)$ is the maximal structure that models $\widehat{r}(F)$; therefore, it must be that $coerce_{\widehat{r}(F)}(S) = T$.

C.3 Local Safety of Shape Analysis

Theorem 6.29 [Local Safety Theorem]. *If vertex w is a condition, then for all $S \in$ 3-STRUCT$[\mathcal{P} \cup \{sm\}]$*

i. If $S^\natural \in \gamma(S)$ and $S^\natural \models cond(w)$, then there exists $S' \in \widehat{coerce}(focus_{F(w)}(S))$ such that $S' \models_3 cond(w)$ and $S^\natural \in \gamma(t\text{-}embed_c(S'))$.

ii. If $S^\natural \in \gamma(S)$ and $S^\natural \models \neg cond(w)$, then there exists $S' \in \widehat{coerce}(focus_{F(w)}(S))$ such that $S' \models_3 \neg cond(w)$ and $S^\natural \in \gamma(t\_embed_c(S'))$.

If vertex w is a statement, then

iii. If $S^\natural \in \gamma(S)$, then $[\![st(w)]\!](S^\natural) \in \widehat{\gamma}(\widehat{t\_embed_c}(\widehat{coerce}(\widehat{[\![st(w)]\!]}_3(focus_{F(w)}(S)))))$.

Proof: Let w be either a statement or condition of the control-flow graph, let S be a structure in 3-STRUCT$[\mathcal{P} \cup \{sm\}]$, and let $S^\natural \in \gamma(S)$. By Defn. 6.4, there exists $S_1 \in focus_{F(w)}(S)$, such that $S^\natural \in \gamma(S_1)$. The proof proceeds as follows.

Let us show that *i.* holds. Assume that $S^\natural \models cond(w)$. By Defn. 4.8 (as modified by footnote 12), $S^\natural \in \gamma(S)$ means that there is a mapping f such that $S^\natural \sqsubseteq^f S_1$ and $S^\natural$ satisfies the compatibility formulae F. Let S_2 be the tight embedding of $S^\natural$ with respect to f. Thus, $S_2 \sqsubseteq S_1$. By Lemma 6.15, $S_2 \models \widehat{r}(F)$. Therefore, by Defn. 6.23, $S_2 \sqsubseteq coerce_{\widehat{r}(F)}(S_1)$. In particular, this means that there is a mapping f' such that $S_2 \sqsubseteq^{f'} coerce_{\widehat{r}(F)}(S_1)$. By the transitivity of embedding, we have $S^\natural \sqsubseteq^{f' \circ f} coerce_{\widehat{r}(F)}(S_1)$. Because $S^\natural \models cond(w)$, the Embedding Theorem implies that $coerce_{\widehat{r}(F)}(S_1) \models_3 cond(w)$. Finally, t_embed_c simply folds together and renames individuals from $\mathit{coerce}_{\widehat{r}(F)}(S_1)$, so we know that the composed mapping $(t\_\mathit{embed}_c \circ f' \circ f)$ embeds $S^\natural$ into $t\_\mathit{embed}_c(\mathit{coerce}_{\widehat{r}(F)}(S_1))$:

$$S^\natural \sqsubseteq^{(t\_\mathit{embed}_c \circ f' \circ f)} t\_\mathit{embed}_c(\mathit{coerce}_{\widehat{r}(F)}(S_1)).$$

By the definition of γ (Defn. 4.8), this implies property *i.* for $S' = \mathit{coerce}_{\widehat{r}(F)}(S_1)$.

The proof of *ii.* is identical to the proof of *i.*, with *cond* replaced by $\neg\mathit{cond}$.

Let us show that *iii.* holds for a statement w. It follows from the Embedding Theorem and the definitions of $[\![st(w)]\!]$ and $[\![st(w)]\!]_3$ (Defs. 3.3 and 3.5) that,

$$\text{If } S^\natural \in \gamma(S_1), \text{ then } [\![st(w)]\!](S^\natural) \in \gamma([\![st(w)]\!]_3(S_1)).$$

In particular, this means that there is a mapping f such that $[\![st(w)]\!]({S_1}^\natural) \sqsubseteq^f [\![st(w)]\!]_3(S_1)$. Let S_2 be the tight embedding of $[\![st(w)]\!](S^\natural)$ with respect to f. Thus, $S_2 \sqsubseteq [\![st(w)]\!]_3(S_1)$. Because $[\![st(w)]\!](S^\natural) \models F$, Lemma 6.15 implies that $S_2 \models \widehat{r}(F)$. Therefore, by Defn. 6.23, $S_2 \sqsubseteq \mathit{coerce}_{\widehat{r}(F)}([\![st(w)]\!]_3(S_1))$. In particular, this means that there is a mapping f' such that $S_2 \sqsubseteq^{f'} \mathit{coerce}_{\widehat{r}(F)}([\![st(w)]\!]_3(S_1))$. By the transitivity of embedding, we have $[\![st(w)]\!](S^\natural) \sqsubseteq^{f' \circ f} \mathit{coerce}_{\widehat{r}(F)}([\![st(w)]\!]_3(S_1))$. However, t_embed_c simply folds together and renames individuals from $\mathit{coerce}_{\widehat{r}(F)}([\![st(w)]\!]_3(S_1))$, so we know that the composed mapping $(t\_\mathit{embed}_c \circ f' \circ f)$ embeds $[\![st(w)]\!](S^\natural)$ into $t\_\mathit{embed}_c(\mathit{coerce}_{\widehat{r}(F)}([\![st(w)]\!]_3(S_1)))$:

$$[\![st(w)]\!](S^\natural) \sqsubseteq^{(t\_\mathit{embed}_c \circ f' \circ f)} t\_\mathit{embed}_c(\mathit{coerce}_{\widehat{r}(F)}([\![st(w)]\!]_3(S_1))).$$

By the definition of γ (Defn. 4.8), this implies property *iii.*