

# Logical Characterizations of Heap Abstractions

GRETA YORSH

Tel-Aviv University

THOMAS REPS

University of Wisconsin

MOOLY SAGIV

Tel-Aviv University

and

REINHARD WILHELM

Universität des Saarlandes

---

Shape analysis concerns the problem of determining “shape invariants” for programs that perform destructive updating on dynamically allocated storage. In recent work, we have shown how shape analysis can be performed using an abstract interpretation based on three-valued first-order logic. In that work, concrete stores are finite two-valued logical structures, and the sets of stores that can possibly arise during execution are represented (conservatively) using a certain family of finite three-valued logical structures. In this article, we show how three-valued structures that arise in shape analysis can be characterized using formulas in first-order logic with transitive closure. We also define a nonstandard (“supervaluational”) semantics for three-valued first-order logic that is more precise than a conventional three-valued semantics, and demonstrate that the supervaluational semantics can be implemented using existing theorem provers.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification

General Terms: Verification

Additional Key Words and Phrases: Logic, characterization, canonical abstraction, shape analysis

## ACM Reference Format:

Yorsh, G., Reps, T., Sagiv, M., and Wilhelm, R. 2007. Logical characterizations of heap abstractions. *ACM Trans. Comput. Logic* 8, 1, Article 5 (Jan. 2007), 27 pages. DOI = 10.1145/1182613.1182618 <http://doi.acm.org/10.1145/1182613.1182618>

---

Authors' addresses: G. Yorsh, School of Computer Science, Tel-Aviv University, P.O. Box 39040, Tel Aviv 69978, Israel; email: gretay@post.tau.ac.il; T. Reps, Computer Science Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706-1685; M. Sagiv, School of Computer Science, Tel-Aviv University, P.O. Box 39040, Tel Aviv 69978, Israel; R. Wilhelm, Informatik, Universität des Saarlandes, Rechenzentrum, Postfach 15 11 50, 66041 Saarbrücken, Germany.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org). © 2007 ACM 1529-3785/2007/01-ART5 \$5.00 DOI 10.1145/1182613.1182618 <http://doi.acm.org/10.1145/1182613.1182618>

## 1. INTRODUCTION

Abstraction and abstract interpretation [Cousot and Cousot 1977] are key tools for automatically verifying properties of systems, both for hardware systems [Clarke et al. 1994; Dams 1996] and software systems [Nielson et al. 1999]. In abstract interpretation, sets of concrete stores are represented in a conservative manner by abstract values (as explained to follow). Each transition of the system is given an interpretation over abstract values that is conservative with respect to its interpretation over corresponding sets of concrete stores; that is, the result of “executing” a transition must be an abstract value which describes a superset of the concrete stores which actually arise. This methodology guarantees that the results of abstract interpretation overapproximate the sets of concrete stores which actually arise at each point in the system.

One issue that arises when abstraction is employed concerns the *expressiveness* of the abstraction method: What collections of concrete states can be expressed exactly using the given abstraction method? A second issue that arises is how to *extract information* from an abstract value. For instance, this is a fundamental problem for clients of abstract interpretation, such as verification tools, program optimizers, program-understanding tools, etc., which need to be able to interpret what an abstract value means. An abstract value  $a$  represents a set of concrete stores  $X$ ; ideally, a query  $\varphi$  should return an answer that summarizes the result of posing  $\varphi$  against each concrete store  $S \in X$ :

- If  $\varphi$  is true for each  $S$ , the summary answer should be “true.”
- If  $\varphi$  is false for each  $S$ , the summary answer should be “false.”
- If  $\varphi$  is true for some  $S \in X$ , but false for some  $S' \in X$ , the summary answer should be “unknown.”

This article presents results on both of these questions, for a class of abstractions that originally arose in work on the problem of shape analysis [Jones and Muchnick 1981; Chase et al. 1990; Sagiv et al. 2002]. Shape analysis concerns the problem of finding “shape descriptors” that characterize shapes of the data structures that a program’s pointer variables point to. Shape analysis is one of the most challenging problems in abstract interpretation because it generally deals with programs written in languages like C, C++, and Java, which allow: (i) dynamic allocation and deallocation of cells from the heap, (ii) destructive updating of structure fields, and, in the case of Java, (iii) dynamic creation and destruction of threads. This combination of features creates considerable difficulties for any abstract-interpretation method.

The motivation for the present article was to understand the expressiveness of the shape abstractions defined in Sagiv et al. [2002]. In that work, concrete stores are finite two-valued logical structures, and the sets of stores that can possibly arise during execution are represented (conservatively) using a certain family of finite three-valued logical structures. In this setting, an abstract value is a set of three-valued logical structures.

Because the notion of abstraction used in Sagiv et al. [2002] is based on logical structures, our results are actually more broadly applicable than shape-analysis problems. For example, it was applied to verification of sorting

algorithms [Lev-Ami et al. 2000]; showing absence of concurrent modification exception [Ramalingam et al. 2002]; correct usage of JDBC, I/O streams, Java collections and iterators [Yahav and Ramalingam 2004]; correctness of concurrent queue algorithms [Yahav and Sagiv 2003]; modeling concurrency in Java programs which contain dynamic creation of objects and threads [Yahav 2001]; analyzing processes in ambient calculus [Nielson et al. 2000]; and reducing space consumption of Java programs via compile-time memory management, with application to JavaCard programs [Shaham et al. 2003].

In fact, our results apply to any abstraction in which concrete states of a system are represented by finite two-value logical structure and abstraction is performed via the mechanisms described in Sections 2 and 3. The approach taken in the article should also be relevant for addressing expressibility issues for a number of other abstractions that are related to Sagiv et al. [2002], including McMillan [1999], Kuncak et al. [2002], Godefroid and Jagadeesan [2003], Huth et al. [2001], and Clarke et al. [1994, 2000], as well as for the *allocation-site* abstraction—often used in points-to analysis [Andersen 1993; Steensgaard 1996; Shapiro and Horwitz 1997; Fähndrich et al. 1998; Su et al. 2000; Das 2000; Heintze and Tardieu 2001]—in which all objects allocated at a single statement are represented by a single “abstract memory object” [Jones and Muchnick 1982; Chase et al. 1990]. Throughout the article, however, we use shape-analysis examples to illustrate the concepts discussed.

The article investigates the expressiveness of finite three-valued structures by giving a logical characterization of these structures; that is, we examine the question

For a given three-valued structure  $S$ , under what circumstances is it possible to create a formula  $\hat{\gamma}(S)$  such that  $S^\natural$  satisfies  $\hat{\gamma}(S)$  exactly when  $S^\natural$  is a two-valued structure that  $S$  represents? In other words,  $S^\natural \models \hat{\gamma}(S)$  iff  $S$  represents  $S^\natural$ .

This article presents two results concerning this question:

- It is not possible to give a formula  $\hat{\gamma}(S)$  written in first-order logic with transitive closure for an arbitrary structure  $S$  (unless  $NL = NP$ , see Section 3). However, this is always possible for a well-defined class of three-valued structures (this class includes all the three-valued structures that have been shown to be useful for shape analysis [Sagiv et al. 2002]).
- Moreover, it is always possible to give a  $\hat{\gamma}(S)$  in general, using a more powerful formalism, namely, monadic second-order formulas.

The ability to write a formula  $\hat{\gamma}(S)$  that exactly captures what  $S$  represents provides a fundamental tool for improving TVLA [Lev-Ami and Sagiv 2000] through the use of symbolic methods. The current TVLA system performs iterative fixed-point computations and yields, at every program point, a set of three-valued structures which represent a superset of all possible stores that can arise at this point in any execution. However, TVLA suffers from two limitations: (i) It is not always as precise as possible (as explained to follow); (ii) it does not scale to handle large programs because the worst-case complexity of

the algorithm is doubly exponential in certain parameters (typically, the number of program variables).

The contributions of this article lay the required groundwork for using symbolic techniques to address both of these limitations. The ability to characterize a three-valued structure  $S$  by a formula  $\widehat{\gamma}(S)$  is a key step toward harnessing a standard (two-valued) theorem prover to aid in abstract interpretation:

- computing the effect of a program statement on an abstract value in the most precise way possible for a given shape-analysis abstraction; and
- developing a modular shape-analysis by using *assume-guarantee* reasoning. The idea is to allow arbitrary first-order formulas to be used to express pre- and postconditions, thereby enabling the code of each procedure to be analyzed once for all potential contexts. This allows the use of shape analysis for applications in which not all the source code is available. This becomes specifically profitable for recursive procedures, since it avoids the need to iterate shape analysis.

These methods are the subject of Yorsh et al. [2004] and Lam et al. [2005].

Another contribution of this article directly addresses the first of the aforementioned limitations of TVLA's current technique. We give a procedure for extracting information from a three-valued logical structure  $S$  in the most precise way possible. In other words, we give a nonstandard way to check if a formula  $\varphi$  holds in  $S$ :

- If  $\widehat{\gamma}(S) \Rightarrow \varphi$  is valid, that is, holds in all two-valued structures, we know that  $\varphi$  evaluates to 1 in all the two-valued structures represented by  $S$ .
- If  $\widehat{\gamma}(S) \Rightarrow \neg\varphi$  is valid, we know that  $\varphi$  evaluates to 0 in all the two-valued structures represented by  $S$ .
- Otherwise, we know that there exists a two-valued structure represented by  $S$  where  $\varphi$  evaluates to 1, and there exists another two-valued structure represented by  $S$  where  $\varphi$  evaluates to 0.

This method represents the most precise way of extracting information from a three-valued logical structure; in particular, whenever this method returns 1/2 (standing for “unknown”), any sound method for extracting information from  $S$  must also return 1/2. This is in contrast with the techniques used in Sagiv et al. [2002], which can return 1/2 even when all the two-valued structures represented by  $S$  have the value 1 (or all have the value 0).

For practical purposes, the success of using symbolic methods depends on having a terminating theorem prover. Although the validity question is undecidable for first-order logic with transitive closure, several theorem provers for first-order logic have been created. In this article, we report on two experiments in which we used these tools to implement symbolic procedures for extracting information from a three-valued structure in the most precise way possible. We also performed several successful experiments with other symbolic operations [Yorsh et al. 2004; Erez 2004]. Although these experiments are rather preliminary, we believe that this approach can be made to work in practice. For example, there has been some progress recently in using SPASS, including the use of transitive closure [Lev-Ami et al. 2005]. Also, in Immerman et al. [2004a],

```

/* 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->data = d;
    e = y->n;
    t->n = e;
    y->n = t;
}

```

(a)
(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.

we have identified a decidable subset of first-order logic with transitive closure that is useful for shape analysis. We define conditions under which  $\widehat{\gamma}$  can be expressed in this logic (Section 5.2). We are also investigating other decidable logics, as well.

The remainder of the work is organized as follows. Section 2 defines our terminology, and explains the use of three-valued structures as abstractions of two-valued structures. Section 3 presents results on the expressiveness of three-valued structures, and gives an algorithm for generating  $\widehat{\gamma}$  for certain families of three-valued structures. Section 4 discusses the problem of reading out information from a three-valued structure in the most precise way possible. Section 5 discusses the applications of  $\widehat{\gamma}$  to program analysis and some implementation issues. Section 6 discusses related work. Appendix A defines an alternative abstract domain for shape analysis, based on canonical abstraction, and the  $\widehat{\gamma}$  operation for this domain. Appendix B shows how to characterize general three-valued structures. Appendix C contains the details for one of the article’s examples. The proofs appear in Appendix D.

## 2. PRELIMINARIES

Section 2.1 defines the syntax and standard Tarskian semantics of first-order logic with transitive closure and equality. Section 2.2 introduces *integrity formulas*, which exclude structures that do not represent a potential store. Section 2.3 introduces three-valued logical structures which extend ordinary logical structures with an extra value,  $1/2$ , representing “unknown” values that arise when several concrete nodes are represented by a single abstract node. The powerset of three-valued structures forms an abstract domain which is related to the concrete domain consisting of the powerset of two-valued structures via *embedding*, as described in Section 2.4.

Figure 1(a) shows the declaration of a linked-list data type in C, and Figure 1(b) shows a C program that searches a list and splices a new element into the list. This program will be used as a running example throughout this article.

Table I. The Set of Predicates

Predicate	Intended Meaning
$eq(v_1, v_2)$	Do $v_1$ and $v_2$ denote the same heap node?
$q(v)$	Does pointer variable $q$ point to node $v$ ?
$n(v_1, v_2)$	Does the $n$ field of $v_1$ point to $v_2$ ?
$is(v)$	Is $v$ pointed to by more than one field ?
$r_q(v)$	Is the node $v$ reachable from $q$ ?

A set of predicates is for representing the stores manipulated by programs that use the `List` data type from Figure 1(a). Here  $q$  denotes an arbitrary predicate in the set  $PVar$ , which contains a predicate for each program variable of type `List`. In the case of `insert`,  $PVar = \{x, y, t, e\}$ .

## 2.1 Syntax and Semantics of First-Order Formulas with Transitive Closure

We represent concrete stores by ordinary two-valued logical structures over a fixed finite set of predicate symbols  $\mathcal{P} = \{eq, p_1, \dots, p_n\}$ , where  $eq$  is a designated binary predicate denoting equality of nodes. We also use  $maxR$  to denote the maximal arity of the predicates in  $\mathcal{P}$ . Without loss of generality, we exclude constant and function symbols from the logic.<sup>1</sup>

*Example 2.1.* Table I lists the set of predicates used in the running example. 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. The unary predicate  $is$  (“is shared”) captures “heap sharing”, that is, `List` elements pointed to by more than one field (this was introduced in Chase et al. [1990] to capture list and tree data structures). The unary predicates  $r_x$ ,  $r_y$ ,  $r_t$ , and  $r_e$  hold for heap nodes reachable from the program variables  $x$ ,  $y$ ,  $t$ , and  $e$ , respectively. A heap node  $u$  is said to be *reachable* from a program variable if the variable points to a heap node  $u'$ , and it is possible to go from  $u'$  to  $u$  by following zero or more  $n$ -links. Reachability is defined in terms of the reflexive transitive closure of the predicate  $n$ .

The notion of reachability plays a crucial role in defining abstractions that are useful for proving program properties in practice. For instance, it may have the effect of preventing disjoint lists from being collapsed in the abstract representation. This may significantly improve the precision of the answers obtained by a program analysis.

We define first-order formulas inductively over the *vocabulary*  $\mathcal{P}$  using the logical connectives  $\vee$  and  $\neg$ , the quantifier  $\exists$ , and the operator ‘ $TC$ ’ in the standard way:

$$\varphi ::= \mathbf{0} \mid \mathbf{1} \mid p(v_1, \dots, v_k) \mid (\neg\varphi_1) \mid (\varphi_1 \vee \varphi_2) \mid (\exists v_1 : \varphi_1) \mid (TC \ v_1, v_2 : \varphi_1)(v_3, v_4)$$

where  $p \in \mathcal{P}$ ;  $v_i$  are variables;  $\varphi, \varphi_i$  are formulas.

The set of free variables of a formula is defined as usual. A formula is *closed* when it has no free variables. The operator ‘ $TC$ ’ denotes transitive closure. If

<sup>1</sup>Constant symbols can be encoded via unary predicates, and  $n$ -ary functions via  $(n + 1)$ -ary predicates.

$\varphi_1$  is a formula with free variables  $V$ , then  $(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)$  is a formula with free variables  $(V - \{v_1, v_2\}) \cup \{v_3, v_4\}$ .

We use several shorthand notations:  $\varphi_1 \Rightarrow \varphi_2 \stackrel{\text{def}}{=} (\neg\varphi_1 \vee \varphi_2)$ ;  $\varphi_1 \wedge \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2)$ ;  $\varphi_1 \Leftrightarrow \varphi_2 \stackrel{\text{def}}{=} (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$ ; and  $\forall v : \varphi \stackrel{\text{def}}{=} \neg\exists v : \neg\varphi$ . The transitive closure of a binary predicate  $p$  is  $p^+(v_3, v_4) \stackrel{\text{def}}{=} (TC\ v_1, v_2 : p(v_1, v_2))(v_3, v_4)$ . The reflexive transitive closure of a binary predicate  $p$  is  $p^*(v_3, v_4) \stackrel{\text{def}}{=} ((TC\ v_1, v_2 : p(v_1, v_2))(v_3, v_4) \vee eq(v_3, v_4))$ . The order of precedence among the connectives, from highest to lowest, is as follows:  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $'TC'$ ,  $\forall$ , and  $\exists$ . We drop parentheses wherever possible, except for emphasis.

*Definition 2.1 (Two-valued Logical Structures).* Let  $\mathcal{P}_i$  denote the set of predicate symbols with arity  $i$ . A logical structure over  $\mathcal{P}$  is a pair  $S = \langle U, \iota \rangle$  in which

- $U$  is a (possibly infinite) set of nodes.
- $\iota$  is the interpretation of predicate symbols, that is, for every predicate symbol  $p \in \mathcal{P}_i$ ,  $\iota(p) : U^i \rightarrow \{0, 1\}$  determines the tuples for which  $p$  holds. Also,  $\iota(eq)$  is the interpretation of equality, that is,  $\iota(eq)(u_1, u_2) = 1$  iff  $u_1 = u_2$ .

Next, we define the standard Tarskian semantics for first-order logic.

*Definition 2.2 (Semantics of First-Order Logical Formulas).* Consider a logical structure  $S = \langle U, \iota \rangle$ . An assignment  $Z$  is a function that maps free variables to nodes (i.e., an assignment has the functionality  $Z : \{v_1, v_2, \dots\} \rightarrow U$ ). An assignment that is defined on all free variables of a formula  $\varphi$  is called *complete* for  $\varphi$ . In the sequel, we assume that every assignment  $Z$  which arises in connection with the discussion of some formula  $\varphi$  is complete for  $\varphi$ . We say that  $S$  and  $Z$  *satisfy* a formula  $\varphi$  (denoted by  $S, Z \models \varphi$ ) when one of the following holds:

- $\varphi \equiv 1$ .
- $\varphi \equiv p(v_1, v_2, \dots, v_i)$  and  $\iota(p)(Z(v_1), Z(v_2), \dots, Z(v_i)) = 1$ .
- $\varphi \equiv \neg\varphi_0$  and  $S, Z \models \varphi_0$  does not hold.
- $\varphi \equiv \varphi_1 \vee \varphi_2$ , and either  $S, Z \models \varphi_1$  or  $S, Z \models \varphi_2$ .
- $\varphi \equiv \exists v_1 : \varphi_1$  and there exists a node  $u \in U$ ,  $m \geq 2$  such that  $S, Z[v_1 \mapsto u] \models \varphi_1$ .
- $\varphi \equiv (TC\ v_1, v_2 : \varphi_1)(v_3, v_4)$  and there exists  $u_1, u_2, \dots, u_m \in U$ ,  $m \geq 2$  such that  $Z(v_3) = u_1, Z(v_4) = u_m$  and for all  $1 \leq i < m$ ,  $S, Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models \varphi_1$ .

For a closed formula  $\varphi$ , we will omit the assignment in the satisfaction relation, and merely write  $S \models \varphi$ .

## 2.2 Integrity Formula

Because not all logical structures represent stores, we use a designated closed formula  $F$ , called the integrity formula,<sup>2</sup> to exclude structures that are not of

<sup>2</sup>In Sagiv et al. [2002] these are called “hygiene conditions.”

interest; in our application, such structures are those that do not correspond to possible stores. This allows us to restrict the set of structures to those satisfying  $F$ .

*Definition 2.3.* A structure  $S$  is *admissible* if  $S \models F$ .

In the rest of the article, we assume that we work with a fixed integrity formula  $F$ . All our notations are parameterized by  $\mathcal{P}$  and  $F$ .

*Example 2.2.* For the `List` data type, there are four conditions that define the admissible structures. At any time during execution,

- (a) each program variable can point to, at most, one heap node;
- (b) the `n` field of a heap node can point to, at most, one heap node;
- (c) predicate *is* (“is shared”) holds for exactly those nodes that have two or more predecessors; and
- (d) the reachability predicate for each variable  $q$  holds for exactly those nodes that are reachable from program variable  $q$ .

The set  $PVar$  contains a predicate for each program variable of type `List`; in the case of `insert`,  $PVar = \{x, y, t, e\}$ . Thus, the integrity formula  $F_{List}$  for the `List` data type is:

$$\begin{aligned}
& \bigwedge_{p \in PVar} \forall v_1, v_2 : p(v_1) \wedge p(v_2) \Rightarrow eq(v_1, v_2) \quad (a) \\
\wedge & \quad \forall v, v_1, v_2 : n(v, v_1) \wedge n(v, v_2) \Rightarrow eq(v_1, v_2) \quad (b) \\
\wedge & \quad \forall v : is(v) \iff \exists v_1, v_2 : \neg eq(v_1, v_2) \wedge n(v_1, v) \wedge n(v_2, v) \quad (c) \\
\wedge & \quad \bigwedge_{q \in PVar} \forall v : r_q(v) \iff \exists v_1 : q(v_1) \wedge n^*(v_1, v) \quad (d)
\end{aligned}$$

### 2.3 Three-Valued Logical Structures and Embedding

In this section, we define three-valued logical structures, which provides a way to represent a set of two-valued logical structures in a compact and conservative way.

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  $l_1$  possibly has more definite information than  $l_2$ .

*Definition 2.4 (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  $l_2 = 1/2$ .

*Definition 2.5.* A *three-valued logical structure* over  $\mathcal{P}$  is the generalization of the two-valued structures given in Definition 2.1 in that predicates may have the value  $1/2$ . This means that  $S = \langle U, \iota \rangle$ , where for  $p \in \mathcal{P}_i$ ,  $\iota(p) : (U^S)^i \rightarrow \{0, 1, 1/2\}$ . In addition: (i) for all  $u \in U^S$ ,  $\iota^S(eq)(u, u) \sqsupseteq 1$ , and (ii) for all  $u_1, u_2 \in U^S$  such that  $u_1$  and  $u_2$  are distinct nodes,  $\iota^S(eq)(u_1, u_2) = 0$ .

A node  $u \in U$  having  $\iota^S(eq)(u, u) = 1/2$  is called a *summary node*. As we shall see, such a node may represent more than one node from a given two-valued structure.

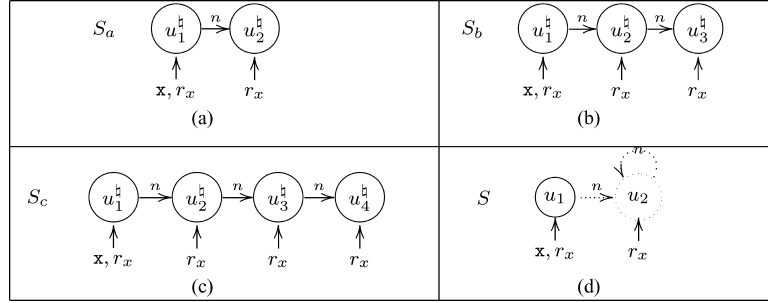


Fig. 2. (a),(b),(c) Examples of two-valued structures representing linked-lists that are pointed to by program variable  $x$ , of length 2, 3, and 4, respectively. (d)  $S$  represents all lists that are pointed to by program variable  $x$  and that have at least two elements, including the lists represented by (a)–(c).

We denote the set of two-valued logical structures by  $2\text{-STRUCT}[\mathcal{P}]$ . The set of three-valued logical structures is denoted by  $3\text{-STRUCT}[\mathcal{P}]$ .

A three-valued structure can be depicted as a directed graph, with nodes as graph nodes. A unary predicate  $p$  is represented in the graph by having a solid arrow from the predicate name  $p$  to node  $u$  for each node  $u$  for which  $\iota(p)(u) = 1$ . An arrow between two nodes indicates whether a binary predicate holds for the corresponding pair of nodes. An indefinite value of a predicate is shown by a dotted arrow; the value 1 is shown by a solid arrow; and the value 0 is shown by the absence of an arrow.

*Example 2.3.* Figure 2(d) shows a three-valued structure that represents possible inputs of the `insert` program. This structure represents all lists that are pointed to by program variable  $x$  and have at least two elements. The structure has two nodes,  $u_1$  and  $u_2$ , where  $u_1$  is the head of the list pointed to by  $x$ , and  $u_2$  is a summary node (drawn as a double circle), which represents the tail of the list. Predicate  $r_x$  holds for  $u_1$  and  $u_2$ , indicating that all elements of the list are reachable from  $x$ . Other unary predicates are not shown, indicating that their values are 0 for all nodes, that is, the program variables  $y$ ,  $e$ , and  $t$  are NULL, and there is no sharing in the list. The dotted edge from  $u_1$  to  $u_2$  indicates that there may be  $n$ -links from the head of the list to some elements in the tail. In fact, the  $(u_1, u_2)$ -edge represents exactly one  $n$ -link that points to exactly one list element because of conjunct (b) of the integrity formula Example 2.2. In contrast, the dotted self-loop on  $u_2$  represents all  $n$ -links that may occur in the tail.

## 2.4 Embedding Order

We define the *embedding ordering* on structures as follows.

*Definition 2.6.* Let  $S = \langle U^S, \iota^S \rangle$  and  $S' = \langle U^{S'}, \iota^{S'} \rangle$  be two logical structures, and let  $f : U^S \rightarrow U^{S'}$  be a surjective. We say that  $f$  *embeds*  $S$  in  $S'$  (denoted by  $S \sqsubseteq^f S'$ ) if for every predicate symbol  $p \in \mathcal{P}_i$  and all  $u_1, \dots, u_i \in U^S$ ,

$$\iota^S(p)(u_1, \dots, u_i) \sqsubseteq \iota^{S'}(p)(f(u_1), \dots, f(u_i)). \quad (1)$$

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

*Example 2.4.* Figure 2(a)–(c) show some of the two-valued structures that can be embedded into the three-valued structure  $S$  shown in Figure 2(d). The function that embeds  $S_a$  into  $S$  maps the node  $u_i^\natural \in U^{S_a}$  to  $u_i \in U^S$ , for  $i = 1, 2$ . The function that embeds  $S_b$  into  $S$  maps the node  $u_1^\natural \in U^{S_b}$  to  $u_1 \in U^S$ , and both  $u_2^\natural, u_3^\natural \in U^{S_b}$  to  $u_2 \in U^S$ . Also, Eq. (1) holds because whenever a predicate has a definite value in  $S$ , the corresponding predicate in  $S_b$  has the same value. For example,  $\iota^S(x)(u_2)$  is 0 and  $f(u_2^\natural) = f(u_3^\natural) = u_2$ , and both  $\iota^{S_b}(x)(u_2^\natural)$  and  $\iota^{S_b}(x)(u_3^\natural)$  are 0. Similarly,  $\iota^S(r_x)(u_2) = 1$ , and both  $\iota^{S_b}(r_x)(u_2^\natural)$  and  $\iota^{S_b}(r_x)(u_3^\natural)$  are 1. For a binary predicate,  $\iota^S(n)(u_2, u_1) = 0$ , and both  $\iota^{S_b}(n)(u_2^\natural, u_1^\natural)$  and  $\iota^{S_b}(n)(u_3^\natural, u_1^\natural)$  are 0.

*Remark.* Embedding can be viewed as a variant of homomorphism [Hell and Nesetril 2004]. In cases where  $S$  is a two-valued structure (i.e., all predicates in  $S$  have definite values, including  $eq$ , which is interpreted as standard equality), checking whether a two-valued structure  $S'$  embeds into  $S$  is equivalent to checking whether there is an isomorphism between  $S'$  and  $S$ . In cases where all nodes in  $S$  are summary nodes (i.e., for all  $u \in U^S$ ,  $\iota^S(eq)(u, u) = 1/2$ ) and all other values of predicates are definite, embedding is equivalent to strong homomorphism. In cases where all nodes in  $S$  are summary nodes and all other values of predicates are either 0 or  $1/2$ , embedding is equivalent to homomorphism. In all other cases, that is, when a predicate value for some tuple in  $S$  is 1, embedding generalizes the notion of homomorphism.

*Remark.* In Definition 2.6, we require that  $f$  be surjective in order to guarantee that a quantified formula, such as  $\exists v : \varphi$ , has consistent values in two three-valued structures  $S$  and  $S'$  related by embedding. 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 the value of  $S'$  on  $\varphi$  is 1 when  $v$  is assigned to  $u'$ . This would permit there to be structures  $S$  and  $S'$  for which the value of  $\exists v : \varphi$  on  $S$  is 0 but its value on  $S'$  is 1.

*Concretization of three-valued structures.* Embedding allows us to define the (potentially infinite) set of concrete structures that a set of three-valued structures represents:

*Definition 2.7 (Concretization of Three-Valued Structures).* For a set of structures  $X \subseteq 3\text{-STRUCT}[\mathcal{P}]$ , we denote by  $\gamma(X)$  the set of two-valued structures that  $X$  represents, that is,

$$\gamma(X) = \{S^\natural \in 2\text{-STRUCT}[\mathcal{P}] \mid \text{exists } S \in X \text{ such that } S^\natural \sqsubseteq S \text{ and } S^\natural \models F\} \quad (2)$$

Also, for a singleton set  $X = \{S\}$ , we write  $\gamma(S)$  instead of  $\gamma(X)$ .

*Example 2.5.* Example 2.4 shows that  $S_a \sqsubseteq S$ ,  $S_b \sqsubseteq S$ , and  $S_c \sqsubseteq S$  for the two-valued structures in Figures 2(a)–(c); also, the integrity formula is satisfied for  $S_a$ ,  $S_b$ , and  $S_c$ . Therefore,  $S_a$ ,  $S_b$ , and  $S_c$  are in the concretization

of three-valued structure  $S$ :  $S_a, S_b, S_c \in \gamma(S)$ . Note that the indefinite values of predicates in  $S$  allow the corresponding values in  $S_b$  to be either 0 or 1. In particular,  $\iota^S(eq)(u_2, u_2) = 1/2$  reflects the fact that the abstract node  $u_2$  may represent more than one concrete node. Indeed,  $S_b$  contains two nodes,  $u_2^\natural$  and  $u_3^\natural$ , that are represented by  $u_2 \in S$ . Also,  $\iota^S(eq)(u_2^\natural, u_3^\natural) = 0$ , but  $\iota^S(eq)(u_2^\natural, u_2^\natural) = 1$ .

The abstract domain we consider is the powerset of three-valued structures, where the ordering relation  $\sqsubseteq$  is defined as follows: For every two sets of three-valued structures  $X_1$  and  $X_2$ ,  $X_1 \sqsubseteq X_2$  iff for all  $S_1 \in X_1$ , there exists  $S_2 \in X_2$  such that  $S_1$  is embedded into  $S_2$ .

**2.4.1 The Analysis Technique.** The TVLA ([Lev-Ami and Sagiv 2000]) system carries out an abstract interpretation [Cousot and Cousot 1977] to collect a set of structures at each program point  $P$ . This involves finding the least fixed point of a certain set of equations. To ensure termination, the analysis is carried out with respect to a finite abstract domain, that is, the set of different structures is finite. When the fixed point is reached, the structures that have been collected at program point  $p$  describe a superset of all the concrete stores that can occur at  $p$ . To determine whether a query is always satisfied at  $p$ , we check whether it holds in all of the structures that were collected there. Instantiations of this framework are capable of establishing nontrivial properties of programs that perform complex pointer-based manipulations of *a priori* unbounded-size heap-allocated data structures.

### 3. CHARACTERIZING THREE-VALUED STRUCTURES BY FIRST-ORDER FORMULAS

This section presents our results on characterizing three-valued structures using first-order formulas. Given a three-valued structure  $S$ , the question that we wish to answer is whether it is possible to give a formula  $\widehat{\gamma}(S)$  that accepts exactly the set of two-valued structures which  $S$  represents, that is,  $(S)^\natural \models \widehat{\gamma}(S)$  iff  $(S)^\natural \in \gamma(S)$ .

This question has different answers, depending on what assumptions are made. The task of generating a characteristic formula for a three-valued structure  $S$  is challenging because we have to find a formula that identifies when embedding is possible, or, in other words, that is satisfied by exactly those two-valued structures that embed into  $S$ . It is not always possible to characterize an *arbitrary* three-valued structure by a first-order formula, that is, there exists a three-valued structure  $S$  for which there is no first-order formula with transitive closure that accepts exactly the set of two-valued structures  $\gamma(S)$ .

For example, consider the three-valued structure  $S$  shown in Figure 3. The absence of a self-loop on any of the three summary nodes implies that a two-valued structure can be embedded into this structure if and only if it can be colored using three colors (Lemma D.1 in the appendix). It is well-known that there exists no first-order formula, even with transitive closure, that expresses three-colorability of undirected graphs, unless  $P = NP$  (e.g., see Immerman [1999] and Courcelle [1996]). Therefore, there is no first-order formula that accepts exactly the set  $\gamma(S)$ .

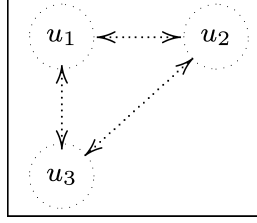


Fig. 3. A three-valued structure that represents three-colorable undirected graphs. A two-valued structure can be embedded into this structure if and only if it can be colored using three colors.

*Remark.* In fact, the condition is even stronger. First-order logic with transitive closure can only express nondeterministic logspace (NL) computations, thus, the NP-complete problem of three-colorability is not expressible in first-order logic unless  $NL = NP$ . This is shown in Immerman [1999] using an ordering relation on the nodes. In our context, without the ordering, the logic is less expressive. Thus, the condition under which three-colorability is expressible is even stronger than  $NL = NP$ . We believe that there is an example of a three-valued structure that is not expressible in the logic, independently of the question of whether  $P = NP$ . However, it is not the main focus of the current article.

### 3.1 FO-Identifiable Structures

Intuitively, the difficulty in characterizing three-valued structures is how to uniquely identify the correspondence between concrete and abstract nodes using a first-order formula. Fortunately, as we will see, for the subclass of three-valued structures used in shape analysis (also known as “bounded structures”), the correspondence can be easily defined using first-order formulas. The bounded structures are a subclass of the three-valued structures in which it is possible to uniquely identify each node using a first-order formula.

*Definition 3.1.* A three-valued structure  $S$  is called *FO-identifiable* if for every node  $u \in U^S$ , there exists a first-order formula  $\text{node}_u^S(w)$  with designated free variable  $w$  such that for every two-valued structure  $S^\natural$  which embeds into  $S$  using a function  $f$ , for every concrete node  $u^\natural \in U^{S^\natural}$ , and for every node  $u_i \in U^S$ :

$$f(u^\natural) = u_i \iff S^\natural, [w \mapsto u^\natural] \models \text{node}_{u_i}^S(w). \quad (3)$$

The idea behind this definition is to have a formula that uniquely identifies each node  $u$  of the three-valued structure  $S$ . This will be used to identify the set of nodes of a two-valued structure that are mapped to  $u$  by embedding. In other words, a concrete node  $u^\natural$  satisfies the *node* formula of, at most, one abstract node, as formalized by the next lemma.

**LEMMA 3.2.** *Let  $S$  be an FO-identifiable structure, and let  $u_1, u_2 \in S$  be distinct nodes. Let  $S^\natural$  be a two-valued structure that embeds into  $S$  and let  $u^\natural \in S^\natural$ . At most, one of the following holds:*

- (1)  $S^\natural, [w \mapsto u^\natural] \models \text{node}_{u_1}^S(w)$
- (2)  $S^\natural, [w \mapsto u^\natural] \models \text{node}_{u_2}^S(w)$

*Remark.* Definition 3.1 can be generalized to handle arbitrary two-valued structures by also allowing extra designated free variables for every concrete node and using equality to check whether the concrete node is equal to the designated variable:  $\text{node}_{u_i}^S(w, v_1, \dots, v_n) \stackrel{\text{def}}{=} w = v_i$ . However, the equality formula cannot be used to identify nodes in a three-valued structure because equality evaluates to  $1/2$  on summary nodes.

We now introduce a standard concept for turning valuations into formulas.

*Definition 3.3.* For a predicate  $p$  of arity  $k$  and truth value  $B \in \{0, 1, 1/2\}$ , we define the formula  $p^B(v_1, v_2, \dots, v_k)$  to be the *characteristic formula of  $B$  for  $p$* , by

$$\begin{aligned} p^0(v_1, v_2, \dots, v_k) &\stackrel{\text{def}}{=} \neg p(v_1, v_2, \dots, v_k) \\ p^1(v_1, v_2, \dots, v_k) &\stackrel{\text{def}}{=} p(v_1, v_2, \dots, v_k) \\ p^{1/2}(v_1, v_2, \dots, v_k) &\stackrel{\text{def}}{=} 1. \end{aligned}$$

The main idea in the preceding definition is that for  $B \in \{0, 1\}$ ,  $p^B$  holds when the value of  $p$  is  $B$ , and for  $B = 1/2$ , the value of  $p$  is unrestricted. This is formalized by the following lemma.

LEMMA 3.4. For every two-valued structure  $S^\natural$  and assignment  $Z$

$$S^\natural, Z \models p^B(v_1, \dots, v_k) \text{ iff } \iota^{S^\natural}(p)(Z(v_1), \dots, Z(v_k)) \sqsubseteq B$$

Definition 3.1 is not a constructive definition because the premises range over arbitrary two-valued structures and arbitrary embedding functions. For this reason, we now introduce a testable condition that implies FO-identifiability.

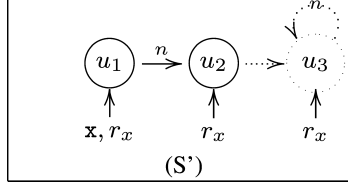
*Bounded structures.* The following subclass of three-valued structures was defined in Sagiv et al. [1999];<sup>3</sup> the motivation there was to guarantee that the shape analysis was carried out with respect to a finite set of abstract structures, and hence that the analysis would always terminate.

*Definition 3.5.* A *bounded structure* over vocabulary  $\mathcal{P}$  is a structure  $S = \langle U^S, \iota^S \rangle$  such that for every  $u_1, u_2 \in U^S$ , where  $u_1 \neq u_2$ , there exists a predicate symbol  $p \in \mathcal{P}_1$  such that: (i)  $\iota^S(p)(u_1) \neq \iota^S(p)(u_2)$  and (ii) both  $\iota^S(p)(u_1)$  and  $\iota^S(p)(u_2)$  are not  $1/2$ , that is,  $\iota^S(p)(u_1), \iota^S(p)(u_2) \in \{0, 1\}$ .

Intuitively, for each pair of nodes in a bounded structure, there is at least one predicate that has different definite values for these nodes. Thus, there is a finite number of different bounded structures (up to isomorphism).

The following lemma shows that bounded structures are FO-identifiable using formulas over unary predicates only (denoted by  $\mathcal{P}_1$ ):

<sup>3</sup>This definition of bounded structures was given in Sagiv et al. [1999]; it is slightly more restrictive than that given in Sagiv et al. [2002] and Lev-Ami [2000], which did not impose requirement 3.5(ii). However, it does not limit the set of problems handled by our method if the structure that is bounded in the weak sense is also FO-identifiable.

Fig. 4. A three-valued structure  $S'$  is FO-identifiable, but not bounded.

LEMMA 3.6. *Every bounded three-valued structure  $S$  is FO-identifiable, where*

$$\text{node}_{u_i}^S(w) \stackrel{\text{def}}{=} \bigwedge_{p \in \mathcal{P}_1} p^{i^S(p)(u_i)}(w) \quad (4)$$

Example 3.1. The first-order *node* formulas for the structure  $S$  shown in Figure 2 are:

$$\begin{aligned} \text{node}_{u_1}^S(w) &= x(w) \wedge r_x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\quad \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \\ \text{node}_{u_2}^S(w) &= \neg x(w) \wedge r_x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\quad \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \end{aligned}$$

*Remark.* In the case that  $S$  is a bounded two-valued structure, the definition of a bounded structure becomes trivial. The reason is that every node in  $S$  can be named by a quantifier-free formula built from unary predicates. This is essentially the same as saying that every node can be named by a constant. If structure  $S'$  embeds into  $S$ , then  $S'$  must be isomorphic to  $S$ , therefore, it is possible to name all nodes of  $S'$  by the same constants. However, this restricted case is not of particular interest for us because to guarantee termination, shape analysis operates on structures that contain summary nodes and indefinite values. In the case that  $S$  contains a summary node, a structure  $S'$  that embeds into  $S$  may have an unbounded number of nodes; hence, the nodes of  $S'$  cannot be named by a finite set of constants in the language.

We already know of interesting cases of FO-identifiable structures that are not bounded, which can be used to generalize the abstraction defined in Sagiv et al. [1999]:

Example 3.2. The three-valued structure  $S'$  in Figure 4 is FO-identifiable by:

$$\begin{aligned} \text{node}_{u_1}^{S'}(w) &\stackrel{\text{def}}{=} x(w) \wedge r_x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\quad \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \\ \text{node}_{u_2}^{S'}(w) &\stackrel{\text{def}}{=} \exists w_1 : x(w_1) \wedge n(w_1, w) \wedge \neg x(w) \wedge r_x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\quad \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \\ \text{node}_{u_3}^{S'}(w) &\stackrel{\text{def}}{=} \neg(\exists w_1 : x(w_1) \wedge n(w_1, w)) \wedge \neg x(w) \wedge r_x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\quad \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \end{aligned}$$

However,  $S'$  is not a bounded structure because nodes  $u_2$  and  $u_3$  have the same values of unary predicates. To distinguish between these nodes, we extended

node $_{u_2}^{S'}(w)$  with the underlined subformula, which captures the fact that only  $u_2$  is directly pointed to by an  $n$ -edge from  $u_1$ .

It can be shown that every FO-identifiable structure can be converted into a bounded structure by introducing more instrumentation predicates. For methodological reasons, we use the notion of FO-identifiable, which directly captures the ability to uniquely identify embedding via (FO) formulas.<sup>4</sup> One of the interesting features of FO-identifiable structures is that the structures generated by a common TVLA operation “focus,” defined in Lev-Ami [2000], are all FO-identifiable (see Lemma D.2 in Appendix D). For example, Figure 4 shows the structure  $S'$ , which is one of the structures resulting from applying the focus operation to the structure  $S$  from Figure 2(d) with the formula  $\exists v_1, v_2 : x(v_1) \wedge n(v_1, v_2)$ . The structure  $S'$  is FO-identifiable, but not bounded. However, structures like the one shown in Figure 3 are not FO-identifiable unless  $P = NP$ .

### 3.2 Characterizing FO-Identifiable Structures

To characterize an FO-identifiable three-valued structure, we must ensure:

- (1) the existence of a surjective embedding function;
- (2) that every concrete node is represented by some abstract node; and
- (3) that corresponding concrete and abstract predicate values meet the embedding condition of Eq. (1).

*Definition 3.7 (First-Order Characteristic Formula).* Let  $S = (U = \{u_1, u_2, \dots, u_n\}, \iota)$  be an FO-identifiable three-valued structure.

We define the *totality characteristic formula* to be the closed formula:

$$\xi_{total}^S \stackrel{\text{def}}{=} \forall w : \bigvee_{i=1}^n \text{node}_{u_i}^S(w) \quad (5)$$

We define the *nullary characteristic formula* to be the closed formula:

$$\xi_{nullary}^S \stackrel{\text{def}}{=} \bigwedge_{p \in \mathcal{P}_0} p^{\iota^S(p)} \quad (6)$$

For a predicate  $p$  of arity  $r \geq 1$ , we define the *predicate characteristic formula* to be the closed formula:

$$\xi^S[p] \stackrel{\text{def}}{=} \forall w_1, \dots, w_r : \bigwedge_{\{u'_1, \dots, u'_r\} \in U} \bigwedge_{j=1}^r \text{node}_{u'_j}^S(w_j) \Rightarrow p^{\iota^S(p)(u'_1, \dots, u'_r)}(w_1, \dots, w_r) \quad (7)$$

<sup>4</sup>In subsequent sections, we redefine this notion to capture other classes of structures.

The *characteristic formula of  $S$*  is defined by:

$$\begin{aligned} \xi^S &\stackrel{\text{def}}{=} \bigwedge_{i=1}^n (\exists v : \text{node}_{u_i}^S(v)) \\ &\wedge \xi_{total}^S \\ &\wedge \xi_{nullary}^S \\ &\wedge \bigwedge_{r=1}^{maxR} \bigwedge_{p \in \mathcal{P}_r} \xi^S[p] \end{aligned} \quad (8)$$

The *characteristic formula of set  $X \subseteq 3\text{-STRUCT}[\mathcal{P}]$*  is defined by:

$$\widehat{\gamma}(X) = F \wedge \left( \bigvee_{S \in X} \xi^S \right) \quad (9)$$

Finally, for a singleton set  $X = \{S\}$ , we write  $\widehat{\gamma}(S)$  instead of  $\widehat{\gamma}(X)$ .

The main ideas behind the four conjuncts of Eq. (8) are:

- The existential quantification in the first conjunct requires that the two-valued structures have at least  $n$  distinct nodes. For each abstract node in  $S$ , the first subformula locates the corresponding concrete node. Overall, this conjunct guarantees that embedding is surjective.
- The totality formula ensures that every concrete node is represented by some abstract node. It guarantees that the embedding function is well-defined.
- The nullary characteristic formula ensures that values of nullary predicates in two-valued structures are at least as precise as those of the corresponding nullary predicates in  $S$ .
- Predicate characteristic formulas guarantee that predicate values in two-valued structures obey the requirements imposed by an embedding into  $S$ .<sup>5</sup>

*Example 3.3.* After a small amount of simplification, the characteristic formula  $\widehat{\gamma}(S)$  for the structure  $S$  shown in Figure 2 is  $F_{List} \wedge \xi^S$ , where  $\xi^S$  is:

$$\begin{aligned} &\exists v : \text{node}_{u_1}^S(v) \wedge \exists v : \text{node}_{u_2}^S(v) \\ &\wedge \forall w : \text{node}_{u_1}^S(w) \vee \text{node}_{u_2}^S(w) \\ &\wedge \bigwedge_{p \in \mathcal{P}_1} \forall w_1 : \bigwedge_{i=1,2} (\text{node}_{u_i}^S(w_1) \Rightarrow p^{i(p)(u_i)}(w_1)) \\ &\wedge \forall w_1, w_2 : (\text{node}_{u_1}^S(w_1) \wedge \text{node}_{u_1}^S(w_2) \Rightarrow eq(w_1, w_2) \wedge \neg n(w_1, w_2) \wedge \neg n(w_2, w_1)) \\ &\quad \wedge (\text{node}_{u_1}^S(w_1) \wedge \text{node}_{u_2}^S(w_2) \Rightarrow \neg eq(w_1, w_2) \wedge \neg n(w_2, w_1)) \end{aligned}$$

The *node<sup>s</sup> Node* formulas are given in Example 3.1, and the predicates for the insert program in Figure 1(b) are shown in Table I. We simplified the formula from Eq. (8) by combining implications that had the same premises. The integrity formula  $F_{List}$  is given in Example 2.2. Note that it uses transitive closure to define the reachability predicates; consequently,  $\widehat{\gamma}(S)$  is a formula in first-order logic with transitive closure.

<sup>5</sup>Definition 3.7 relates to all FO-identifiable structures, not only to bounded structures. For bounded structures, it can be simplified by omitting  $\xi^S[p]$  for all unary predicates  $p$  because it is implied by  $\xi_{total}^S$ . In fact, it can be omitted only for the abstraction predicates, as defined in Sagiv et al. [2002]; however throughout this article, we consider all unary predicates to be abstraction predicates.

When a predicate has an indefinite value on some node tuple, a corresponding conjunct of Eq. (7) can be omitted because it simplifies to 1.

Thus, the size of this simplified version of  $\xi^S$  is linear in the number of definite values of predicates in  $S$ . Assuming that the  $node^S$  formulas contain no quantifiers or transitive-closure operator (e.g., when  $S$  is bounded), the  $\xi^S$  formula has no quantifier alternation and does not contain any occurrences of the transitive-closure operator. Thus, the formula  $\widehat{\gamma}$  is in Existential-Universal normal form (and thus decidable for satisfiability) whenever  $F$  is in Existential-Universal normal form and does not contain transitive closure.<sup>6</sup> Moreover, if the maximal arity of the predicate in  $\mathcal{P}$  is two, then  $\widehat{\gamma}$  is in the two-variable fragment of first-order logic [Mortimer 1975] wherever  $F$  is. In Section 5, we discuss other conditions under which  $\widehat{\gamma}$  can be expressed in a decidable logic.

The following theorem shows that for every FO-identifiable structure  $S$ , the formula  $\widehat{\gamma}(S)$  accepts exactly the set of two-valued structures represented by  $S$ .

**THEOREM 3.8.** *For every FO-identifiable three-valued structure  $S$  and two-valued structure  $S^\natural$ ,  $S^\natural \in \gamma(S)$  iff  $S^\natural \models \widehat{\gamma}(S)$ .*

#### 4. SUPERVALUATIONAL SEMANTICS FOR FIRST-ORDER FORMULAS

In this section, we consider the problem of how to extract information from a three-valued structure by evaluating a query. A compositional semantics for three-valued first-order logic is defined in Sagiv et al. [2002]; however, that semantics is not as precise as the one defined here. The semantics given in this section can be seen as providing the limit of obtainable precision.

*The notion of supervaluational semantics*, defined next, has been used in van Fraassen [1966] and Bruns and Godefroid [2000].

*Definition 4.1 (Supervaluational Semantics of First-Order Formulas).* Let  $X$  be a set of three-valued structures and  $\varphi$  be a closed formula. The *supervaluational semantics of  $\varphi$  in  $X$* , denoted by  $\langle\langle\varphi\rangle\rangle(X)$ , is defined to be the join of the values of  $\varphi$  obtained from each of the two-valued structures that  $X$  represents, that is, the most precise conservative value which can be reported for the value of formula  $\varphi$  in the two-valued structures represented by  $X$  is

$$\langle\langle\varphi\rangle\rangle(X) = \begin{cases} 1 & \text{if } S^\natural \models \varphi \text{ for all } S^\natural \in \gamma(X) \\ 0 & \text{if } S^\natural \not\models \varphi \text{ for all } S^\natural \in \gamma(X) \\ 1/2 & \text{otherwise.} \end{cases} \quad (10)$$

The compositional semantics given in Sagiv et al. [2002] and used in TVLA can yield 1/2 for  $\varphi$ , even when the value of  $\varphi$  is 1 for all the two-valued structures  $S^\natural$  that  $S$  represents (or when the value of  $\varphi$  is 0 for all the  $S^\natural$ ). In contrast, when the supervaluational semantics yields 1/2, we *know* that any sound extraction of information from  $S$  must return 1/2.

*Example 4.1.* We demonstrate now that the supervaluational semantics of the formula  $\varphi_{x \rightarrow \text{next} \neq \text{NULL}} \stackrel{\text{def}}{=} \exists v_1, v_2 : x(v_1) \wedge n(v_1, v_2)$  on the structure  $S$  from

<sup>6</sup>For practical reasons, we often replace the *node* formula by a new (definable) predicate, and add its definition to the integrity formula.

```

procedure Supervaluation( $\varphi$ : Formula,
                        X: Set of 3-valued structures): Value
  if ( $\widehat{\gamma}(X) \Rightarrow \varphi$  is valid) return 1;
  else if ( $\widehat{\gamma}(X) \Rightarrow \neg\varphi$  is valid) return 0;
  otherwise return 1/2;

```

Fig. 5. A procedure for computing the supervaluational value of a formula  $\varphi$  that encodes a query on a three-valued structures  $S$ .

Figure 2(d) is 1. In other words, we wish to argue that for all of the two-valued structures that structure  $S$  from Figure 2(d) represents, the value of the formula  $\varphi_{x \rightarrow \text{next} \neq \text{NULL}}$  must be 1.

We reason as follows:  $S$  represents a list with at least two nodes; that is, all two-valued structures represented by  $S$  have at least two nodes. One node,  $u_1^{\dagger}$ , corresponding to  $u_1$  in  $S$ , is pointed to by program variable  $x$ . The other node, corresponding to the summary node  $u_2$ , must be reachable from  $x$ . Consider the sequence of nodes reachable from  $x$ , starting with  $u_1^{\dagger}$ . Denote the first node in the sequence that embeds into  $u_2$  by  $u_2^{\dagger}$ . By the definition of reachability, there must be an  $n$ -link to  $u_2^{\dagger}$  from a node embedded into  $u_1$ . But the integrity rules guarantee that there is exactly one node that embeds into  $u_1$ , namely,  $u_1^{\dagger}$ . Therefore, the formula  $x(v_1) \wedge n(v_1, v_2)$  holds for  $[v_1 \mapsto u_1^{\dagger}, v_2 \mapsto u_2^{\dagger}]$ .

Note that this formula will be evaluated to 1/2 by TVLA because  $x(v_1) \wedge n(v_1, v_2)$  evaluates to 1/2 under the assignment  $[v_1 \mapsto u_1, v_2 \mapsto u_2]$ : The compositional semantics yields  $x(u_1) \wedge n(u_1, u_2) = 1 \wedge 1/2 = 1/2$ .

Notice that Definition 4.1 does not provide a constructive way to compute  $\llbracket \varphi \rrbracket(X)$  because  $\gamma(X)$  is usually an infinite set.

*Computing supervaluational semantics using theorem provers.* If an appropriate theorem prover is at hand,  $\llbracket \varphi \rrbracket(S)$  can be computed with the procedure shown in Figure 5. This procedure is not an algorithm because the theorem prover might not terminate. Termination can be assured by using standard techniques (e.g., having the theorem prover return a safe answer if a time-out threshold is exceeded) at the cost of losing the ability to guarantee that a most precise result is obtained. If the queries posed by operation Supervaluation can be expressed in a decidable logic, the algorithm for computing supervaluation can be implemented using a decision procedure for this logic. In Section 5, we discuss such decidable logics which are useful for shape analysis.

## 5. APPLICATIONS

The experiments discussed in this section demonstrate how the  $\widehat{\gamma}$  operation can be harnessed in the context of program analysis: The results described to follow go beyond what previous systems were capable of. In Section 5.1, we discuss the use of existing theorem provers and their limitations. In Section 5.2, we suggest a way to overcome these limitations using decidable logic.

We present two examples that use  $\widehat{\gamma}$  to read out information from three-valued structures in a conservative, but rather precise way. The first example demonstrates how supervaluational semantics allows us to obtain more precise information from a three-valued structure than we would using compositional

semantics. The second example demonstrates how to use the three-valued structures obtained from a TVLA analysis to construct a loop invariant; this is then used to show that certain properties of a linked data structure hold on each loop iteration. In addition, we briefly describe how  $\widehat{\gamma}$  can be used in algorithms for computing the most precise abstract operations for shape analysis. Finally, we report on other work that employs  $\widehat{\gamma}$  to generate a concrete counterexample for shape analysis.

*Remark.* The  $\widehat{\gamma}$  operation defines a symbolic concretization with respect to a given abstract domain. In Section 3, we defined  $\widehat{\gamma}$  for the abstract domain of sets of three-valued structures. In Appendix A, we describe a related abstract domain and define  $\widehat{\gamma}$  for this. The applications described in this section can be used with any domain for which  $\widehat{\gamma}$  is defined in some logic and a theorem prover for that logic exists. In our examples, we use  $\widehat{\gamma}$  defined in Section 3 and the first-order logic with transitive closure.

### 5.1 Using the First-Order Theorem Prover SPASS

The TVLA [Lev-Ami and Sagiv 2000] system performs an iterative fixed-point computation which yields at every program point  $p$  a set  $X_p$  of bounded structures. It guarantees that  $\gamma(X_p)$  is a superset of the two-valued structures that can arise at  $p$  in any execution. We have implemented the  $\widehat{\gamma}$  operation in TVLA, and employed SPASS [Weidenbach 2006] to check, using the formula  $\widehat{\gamma}(X_p)$ , that certain properties of the heap hold at program point  $p$ . Also, we implemented the supervalational procedure described in Section 4, employing SPASS. The enhanced version of TVLA generates the formula  $\widehat{\gamma}(S)$  and makes, at most, two calls to SPASS to compute the supervalational value of a query  $\varphi$  in structure  $S$ . In this section, we report on our experience in using SPASS and the problems we have encountered.

First, calls to the SPASS theorem prover need not terminate because first-order logic is undecidable in general. However, in the examples described next, SPASS always terminated.

*Example 5.1.* In Example 4.1 we (manually) proved that the supervalational value of the formula  $\varphi_{x \rightarrow \text{next} \neq \text{NULL}}$  on the structure  $S$  from Figure 2(d) is 1. To check this automatically, we used SPASS to determine the validity of  $\widehat{\gamma}(S) \Rightarrow \varphi_{x \rightarrow \text{next} \neq \text{NULL}}$ ; SPASS indicated that the formula is valid. This guarantees that the formula  $\varphi_{x \rightarrow \text{next} \neq \text{NULL}}$  evaluates to 1 on all of the two-valued structures that embed into  $S$ .

In contrast, TVLA uses Kleene semantics for three-valued formulas, and will evaluate the formula  $\varphi_{x \rightarrow \text{next} \neq \text{NULL}}$  to 1/2: under the assignment  $[v_1 \mapsto u_1, v_2 \mapsto u_2]$ ,  $x(v_1) \wedge n(v_1, v_2)$  evaluates to  $1 \wedge 1/2$ , which equals 1/2.

**5.1.1 Generating and Querying a Loop Invariant.** We used TVLA to compute, for each program point  $p$ , a set  $X_p$  of bounded structures that overapproximate the set of stores that may occur at this point. We then generated  $\widehat{\gamma}(X_p)$ . Because TVLA is sound,  $\widehat{\gamma}(X_p)$  must be an invariant that holds at program point  $p$ , according to Theorem 3.8. In particular, when  $p$  is a program point that begins a loop,  $\widehat{\gamma}(X_p)$  is a loop invariant.

*Example 5.2.* Let  $X = \{S_i \mid i = 1, \dots, 5\}$  denote the set of five three-valued structures that TVLA found at the beginning of the loop in the `insert` program from Figure 2. Table II of Appendix C shows the  $S_i$  and their characteristic formulas. The loop invariant is defined by

$$\widehat{\gamma}(X) = F_{List} \wedge \left( \bigvee_{i=1}^5 \xi^{S_i} \right)$$

Using SPASS, this formula was then used to check that in every structure which can occur at the beginning of the loop,  $x$  points to a valid list, that is, one that is acyclic and unshared. This property is defined by the following formulas:

$$\begin{aligned} \text{acyc}_x &\stackrel{\text{def}}{=} \forall v_1, v_2 : r_x(v_1) \wedge n^+(v_1, v_2) \Rightarrow \neg n^+(v_2, v_1) \\ \text{uns}_x &\stackrel{\text{def}}{=} \forall v : r_x(v) \Rightarrow \neg(\exists w_1, w_2 : \neg eq(w_1, w_2) \wedge n(w_1, v) \wedge n(w_2, v)) \\ \text{list}_x &\stackrel{\text{def}}{=} \text{acyc}_x \wedge \text{uns}_x. \end{aligned}$$

We applied SPASS to check the validity of  $\widehat{\gamma}(S) \Rightarrow \text{list}_x$ ; SPASS indicated that the formula is valid.<sup>7</sup>

In addition to the termination issue, a second obstacle is that SPASS considers infinite structures, which are not allowed in our setting.<sup>8</sup> As a consequence, SPASS can fail to verify that a formula is valid for our intended set of structures; however, the opposite can never happen: Whenever SPASS indicates that a formula is valid, it is indeed valid for our intended set of structures.

*Example 5.3.* We tried to verify that every concrete linked list represented by the three-valued structure  $S$  from Figure 2(d) has a last element. This condition is expressed by the formula  $\varphi_{last} \stackrel{\text{def}}{=} \exists v_1 \forall v_2 : \neg n(v_1, v_2)$ . The supervaluational value of  $\varphi_{last}$  on a structure  $S$  is  $\langle\langle \varphi \rangle\rangle(S) = 1$ , for the following reasons. Because  $r_x$  has the definite value 1 on  $u_2$  in  $S$ , all concrete nodes represented by the summary node  $u_2$  must be reachable from  $x$ . Thus, these nodes must form a linked list, that is, each of these concrete nodes, except for one node that is the “last,” has an  $n$ -edge to another concrete node represented by  $u_2$ . The last node does not have an  $n$ -edge back to any of the nodes represented by  $u_2$  because this would create sharing, whereas the value of predicate  $is$  in  $S$  is 0 on  $u_2$ . Also, the last node cannot have an  $n$ -edge to the concrete node represented by  $u_1$  because the value of predicate  $n$  on the pair  $\langle u_2, u_1 \rangle$  in  $S$  is 0. Therefore, the last element cannot have an outgoing  $n$ -edge.

We used SPASS to determine the validity of  $\widehat{\gamma}(S) \Rightarrow \varphi_{last}$ ; SPASS indicated that the formula is *not* valid because it considered a structure that has infinitely many concrete nodes, all represented by  $u_2$ . Each of these concrete nodes has an  $n$ -edge to the next node.

The validity test of the formula  $\widehat{\gamma}(S) \Rightarrow \neg \varphi_{last}$  failed, of course, because there exists a finite structure that is represented by  $S$  (and thus satisfies  $\widehat{\gamma}(S)$ ) and has a last element. For example, the structure in Figure 2(a) represents a list

<sup>7</sup>SPASS input is available from [www.cs.tau.ac.il/~gretay](http://www.cs.tau.ac.il/~gretay).

<sup>8</sup>Our intended structures are finite because they represent memory configurations which are guaranteed to be finite, although their size is not bounded.

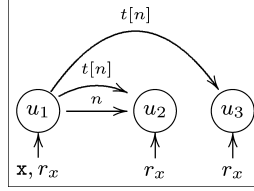


Fig. 6. SPASS takes into account structures in which the  $t[n]$  predicate overapproximates the  $n^+$  predicate, such as the structure shown in this figure.

of size 2. Therefore, the procedure  $Supervaluation(\varphi_{last}, S)$  implemented using SPASS returns  $1/2$ , even though the supervaluational value of  $\varphi_{last}$  on  $S$  is 1.

The third, and most severe, problem that we face is that SPASS does not support transitive closure. Because transitive closure is not expressible in first-order logic, we could only partially model transitive closure in SPASS, as described to follow.

SPASS follows other theorem provers in allowing axioms to express requirements on the set of structures considered. We used SPASS axioms to model integrity rules. To partially model transitive closure, we replaced uses of  $n^+(v_1, v_2)$  by uses of a new designated predicate  $t[n](v_1, v_2)$ . Therefore, SPASS will consider some structures that do not represent possible stores. As a consequence, SPASS can fail to verify that a formula is valid for our intended set of structures; however, the opposite can never happen: Whenever SPASS indicates that a formula is valid, it is indeed valid for our intended set of structures. To avoid some of the spurious failures to prove validity, we added axioms to guarantee that: (i)  $t[n](v_1, v_2)$  is transitive, and (ii)  $t[n](v_1, v_2)$  includes all of  $n(v_1, v_2)$ ; thus,  $t[n](v_1, v_2)$  includes all of  $n^+(v_1, v_2)$ . Because transitive closure requires a minimal set which is not expressible in first-order logic, this approach provides a looser set of integrity rules than we would like. However, it is still the case that whenever SPASS indicates that a formula is valid, it is indeed valid for the set of structures in which  $t[n](v_1, v_2)$  is exactly  $n^+(v_1, v_2)$ .

*Example 5.4.* SPASS takes into account the structure shown in Figure 6, in which the value of  $t[n](u_1, u_3)$  is 1, but the value of  $n^+(u_1, u_3)$  is 0 because there is no  $n$ -edge from  $u_2$  to  $u_3$ .

*Remark.* For practical purposes, the success of using symbolic methods depends on having a terminating theorem prover. We have successfully used SPASS as part of a prototype implementation of the *assume* operation (Section 5.3), and the path-pruning optimization for counterexample generation (Section 5.4). Although these experiments are rather preliminary, we believe that this approach can be made to work in practice. For example, there has been some recent progress in using SPASS, including the use of transitive closure [Lev-Ami et al. 2005]. Also, we have investigated a complementary approach, discussed in Section 5.2.

## 5.2 Decidable Logic

The obstacles mentioned in Section 5.1 are not specific to SPASS. They occur in all theorem provers for first-order logic that we are aware of. To address these

obstacles, we are investigating the use of a decidable logic. To reason about linked data structures, we need a notion of reachability to be expressible, for example, using transitive closure. However, a logic that is both decidable and includes reachability must be limited in other aspects.

One such example is the decidable second-order theory of two successors *WS2S* [Rabin 1969]; its decision procedure is implemented in a tool called *MONA* [Henriksen et al. 1996]. Second-order quantification suffices to express reachability, but there are still two problems. First, the decision procedure for *WS2S* is necessarily nonelementary [Meyer 1975]. Second, *WS2S* only applies to trees, or equivalently, to function graphs (graphs with, at most, one edge leaving any vertex).

Another example is  $EA(TC, f^1)$ , which is a subset of first-order logic with transitive closure in which the following restrictions are imposed on formulas: (i) They must be in existential-universal form, and (ii) they must use (at most) a single unary function  $f$ , but can use an arbitrary number of unary predicates. Immerman et al. [2004a] shows that the decision procedure for satisfiability of  $EA(TC, f^1)$  is NEXPTIME-complete.

In spite of their limitations, both *WS2S* and  $EA(TC, f^1)$  can be useful for reasoning about shape invariants and mutation operations on data structures, such as singly and doubly linked lists, (shared) trees, and graph types [Klarlund and Schwartzbach 1993]. The key is the *simulation technique* [Immerman et al. 2004b], which encodes complex data structures using *tractable* structures, for example, function graphs or simple trees where we can reason with decidable logics.

For example, given a suitable simulation,  $\hat{\gamma}$  formula can be expressed in *WS2S* and  $EA(TC, f^1)$  if the integrity formula  $F$  can. This follows from the definition of  $\hat{\gamma}$  in Eq. (9) and the fact that  $\xi^S$  does not contain quantifier alternation. This makes  $EA(TC, f^1)$  and *WS2S* candidate implementations for the decision procedure used in both the supervalational semantics and in the algorithms described next.

### 5.3 Assume-Guarantee Shape Analysis

The  $\hat{\gamma}$  operation is useful beyond computing supervalational semantics: It is a necessary operation used in the algorithms described in Yorsh et al. [2004] and Reps et al. [2004]. These algorithms perform abstract operations symbolically by representing abstract values as logical formulas, and use a theorem prover to check the validity of these formulas. These algorithms improve on existing shape-analysis techniques by:

- conducting abstract interpretation in the most precise fashion, improving the technique used in the TVLA system [Lev-Ami and Sagiv 2000; Sagiv et al. 2002], which provides no guarantees about the precision of its basic mechanisms; and
- performing modular verification using assume-guarantee reasoning and procedure specifications. This is perhaps the most exciting potential application of  $\hat{\gamma}$ , and  $EA(TC, f^1)$  logic, because existing mechanisms for shape analysis, including TVLA, do not support assume-guarantee reasoning.

#### 5.4 Counterexample Generation

Some preliminary work using the techniques presented in this article to improve the applicability of TVLA has been carried out. The tool described in Erez et al. [2003] and Erez [2004] uses the  $\hat{\gamma}$  operation to generate a concrete counterexample for a potential error message produced by TVLA for an intermediate three-valued structure  $S$  at a program point  $p$ . Such a tool is useful to check if a reported error is a real error or a false alarm, that is, it never occurs on any concrete store.

Generation of concrete counterexamples from  $S$  proceeds as follows. First,  $S$  is converted to the formula  $\hat{\gamma}(S)$ . Then, the tool uses weakest precondition to generate a formula that represents the stores (at the entry point) that lead to an execution trace which reaches  $p$  with a store that satisfies  $\hat{\gamma}(S)$ . Finally, a separate tool [McCune 2001] generates a concrete store that satisfies the formula for the entry point.

### 6. RELATED WORK

There is a sizeable literature on *structure-description formalisms* for describing properties of linked data structures (see Benedikt et al. [1999] and Sagiv et al. [2002] for references). The motivation for the present article was to understand the expressive power of the shape abstractions defined in Sagiv et al. [2002].

In previous work, Benedikt et al. [1999] showed how to translate two kinds of shape descriptors, namely, “path matrices” [Hendren 1990; Hendren and Nicolau 1990] and the variant of shape graphs, discussed in Sagiv et al. [1998], into a logic called  $L_r$  (“logic of reachability expressions”). The shape graphs from Sagiv et al. [1998] are also amenable to the techniques presented in the present article: The characteristic formula defined in Eq. (8) is much simpler than the translation to  $L_r$  given in Benedikt et al. [1999]; moreover, Eq. (8) applies to a more general class of shape descriptors. However, the logic used in Benedikt et al. [1999] is decidable, which guarantees that terminating procedures can be given for problems that can be addressed using  $L_r$ .

The Pointer Analysis Logic Engine (PALE) [Møller and Schwartzbach 2001] provides a structure-description formalism that serves as an assertion language; assertions are translated to second-order monadic logic and fed to MONA. PALE does not handle all data structures, but can handle data structures describable as graph types [Klarlund and Schwartzbach 1993]. Because the logic used by MONA is decidable, PALE is guaranteed to terminate.

One point of contrast between the shape abstractions based on three-valued structures studied in this article and both  $L_r$  and the PALE assertion language is that the powerset of three-valued structures forms an abstract domain. This means that three-valued structures can be used for program analysis by setting up an appropriate set of equations and finding its fixed point [Sagiv et al. 2002]. In contrast, when PALE is used for program analysis, an invariant must be supplied for each loop.

Other structure-description formalisms in the literature include ADDS [Hendren et al. 1992] and shape types [Fradet and Metayer 1997].

The supervaluational semantics for first-order logic discussed in Section 4 is related to a number of other supervaluational semantics for partial logics and three-valued logics discussed in the literature [van Fraassen 1966; Blamey 2002; Bruns and Godefroid 2000]. Compared to previous work, an innovation of Figure 5 is the use of  $\widehat{\gamma}$  to translate a three-valued structure to a formula. In fact, Figure 5 is an example of a general reductionist strategy for providing a supervaluational evaluation procedure for abstract domains by using existing logics and theorem provers/decision procedures.

A recent work [Kuncak and Rinard 2003a], which is an abbreviated version of a more extensive presentation of the results reported in Kuncak and Rinard [2003b], provides an alternative characterization of three-valued structures using logical formulas, equivalent to the characterization presented in the present work. The present article, which extends and elaborates on the results of Yorsh [2003], unlike Kuncak and Rinard [2003a, 2003b], reports on experience and algorithmic issues in using logical characterization of structures for shape analysis; this material is important because shape analysis is the primary motivation and the intended application of this article, as well as those of Kuncak and Rinard [2003a, 2003b]. Also, Section A.4 of the present article gives a simple semantic argument for the property of closure under negation, shown in Kuncak and Rinard [2003b] using a different formalism. The technical similarities and differences between the two works are described in a note available from [www.cs.tau.ac.il/~gretay](http://www.cs.tau.ac.il/~gretay).

## 7. FINAL REMARKS

In Reps et al. [2004], we discuss how to perform all operations required for abstract interpretation in the most precise way possible (relative to the abstraction in use) if certain primitive operations can be carried out, and if a sufficiently powerful theorem prover is at hand. Chief among the primitive operations that must be available is  $\widehat{\gamma}$ ; thus, the material that has been presented in this work shows how to fulfill the requirements of Reps et al. [2004] for a family of abstractions based on three-valued structures (essentially, those used in our past work [Sagiv et al. 2002] and in the TVLA system [Lev-Ami and Sagiv 2000]).

In ongoing research, we are investigating the feasibility of actually applying the techniques from Reps et al. [2004] to perform abstract interpretation for abstractions based on three-valued structures. This approach could be more precise than TVLA because, for instance, it would take into account in a first-class way the integrity formula of the abstraction. In contrast, in TVLA, some operations temporarily ignore the integrity formula, and rely on later clean-up steps to rectify matters.

Another step can be taken in this direction, which is to eliminate the use of three-valued structures, and directly carry out fixed-point computations over logical formulas.

We are also investigating the feasibility of using the results from this article to develop a more precise and modular version of TVLA by using assume-guarantee reasoning [Yorsh et al. 2004]. The idea is to allow arbitrary first-order

formulas with transitive closure to be used to express pre- and postconditions, and to analyze the code for each procedure separately.

## ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

## ACKNOWLEDGMENTS

We thank Neil Immerman, Viktor Kuncak, Tal Lev-Ami, and Alexander Rabinovich for their contributions to this article.

## REFERENCES

- ANDERSEN, L. O. 1993. Binding-Time analysis and the taming of C pointers. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, D. Schmidt, Ed. 47–58.
- BENEDIKT, M., REPS, T., AND SAGIV, M. 1999. A decidable logic for describing linked data structures. In *Proceedings of the European Symposium On Programming*. 2–19.
- BLAMEY, S. 2002. Partial logic. In *Handbook of Philosophical Logic*, 2nd. ed., vol. 5, D. Gabbay and F. Guenther, Eds. Kluwer Academic. 261–353.
- BRUNS, G. AND GODEFROID, P. 2000. Generalized model checking: Reasoning about partial state spaces. In *Proceedings of the Concurrency Theory: 11th International Conference (CONCUR)*. Lecture Notes in Computer Science, vol. 1877. Springer Verlag, 168–182.
- CHASE, D., WEGMAN, M., AND ZADECK, F. 1990. Analysis of pointers and structures. In *SIGPLAN Conference on Programming Language Design and Implementation*. 296–310.
- CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2000. Counterexample-Guided abstraction refinement. In *Proceedings of the Conference on Computer-Aided Verification*. 154–169.
- CLARKE, E., GRUMBERG, O., AND LONG, D. 1994. Model checking and abstraction. *Trans. Prog. Lang. Syst.* 16, 5, 1512–1542.
- COURCELLE, B. 1996. On the expression of graph properties in some fragments of monadic second-order logic. In *Descriptive Complexity and Finite Models: Proceedings of a DIAMCS Workshop*, N. Immerman and P. Kolaitis, Eds. American Mathematical Society. 33–57.
- 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 *Symposium on Principles of Programming Languages*. 238–252.
- DAMS, D. 1996. Abstract interpretation and partial refinement for model checking. Ph.D. thesis, Technical University of Eindhoven, Eindhoven, The Netherlands.
- DAS, M. 2000. Unification-Based pointer analysis with directional assignments. In *Conference on Programming Language Design and Implementation*. 35–46.
- EREZ, G. 2004. Generating concrete counter examples for arbitrary abstract domains. M.S. thesis, Tel-Aviv University, Tel-Aviv, Israel. In preparation.
- EREZ, G., SAGIV, M., AND YAHAV, E. 2003. Generating concrete counter examples for arbitrary abstract domains. Unpublished manuscript.
- FAGIN, R. 1975. Monadic generalized spectra. *Z. Math. Logik* 21, 89–96.
- FÄHNDRICH, M., FOSTER, J., SU, Z., AND AIKEN, A. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. 85–96.
- FRADET, P. AND METAYER, D. L. 1997. Shape types. In *Proceedings of the Symposium on Principles of Programming Languages*. 27–39.
- GODEFROID, P. AND JAGADEESAN, R. 2003. On the expressiveness of 3-valued models. In *Proceedings of the 4th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*. 206–222.
- HEINTZE, N. AND TARDIEU, O. 2001. Ultra-Fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

- HELL, P. AND NESETRIL, J. 2004. *Graphs and Homomorphisms*. Oxford University Press.
- HENDREN, L. 1990. Parallelizing programs with recursive data structures. Ph.D. thesis, Cornell University, 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 *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. 249–260.
- HENDREN, L. AND NICOLAU, A. 1990. Parallelizing programs with recursive data structures. *IEEE Trans. Parallel Distrib. Syst.* 1, 1 (Jan.), 35–47.
- HENRIKSEN, J., JENSEN, J., JØRGENSEN, M., KLARLUND, N., PAIGE, B., RAUHE, T., AND SANDHOLM, A. 1996. Mona: Monadic second-order logic in practice. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*. 89–110.
- HUTH, M., JAGADEESAN, R., AND SCHMIDT, D. A. 2001. Modal transition systems: A foundation for three-valued program analysis. In *Proceedings of the Programming Languages and Systems: 10th European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 2028. Springer Verlag. 155–169.
- IMMERMAN, N. 1999. *Descriptive Complexity*. Springer Verlag.
- IMMERMAN, N., RABINOVICH, A., REPS, T., SAGIV, M., AND YORSH, G. 2004a. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic*. Lecture Notes in Computer Science, vol. 3210. Springer Verlag.
- IMMERMAN, N., RABINOVICH, A., REPS, T., SAGIV, M., AND YORSH, G. 2004b. Verification via structure simulation. In *Computer Aided Verification*. Lecture Notes in Computer Science, vol. 3114. Springer Verlag.
- 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, 102–131.
- JONES, N. AND MUCHNICK, S. 1982. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Symposium on Principles of Programming Languages*. 66–74.
- KLARLUND, N. AND SCHWARTZBACH, M. 1993. Graph types. In *Symposium on Principles of Programming Languages*. New York, NY, 196–205.
- KUNCAK, V., LAM, P., AND RINARD, M. C. 2002. Role analysis. In *Proceedings of the Conference POPL*. 17–32.
- KUNCAK, V. AND RINARD, M. 2003a. Boolean algebra of shape analysis constraints. In *Proceedings of the Verification Model Checking and Abstract Interpretation*. Lecture Notes in Computer Science, vol. 2937. Springer Verlag. 59–72.
- KUNCAK, V. AND RINARD, M. 2003b. On Boolean algebra of shape analysis constraints. Tech. Rep., MIT, CSAIL. <http://www.mit.edu/~vkuncak/papers/index.html>.
- LAM, P., KUNCAK, V., AND RINARD, M. 2005. Hob: A tool for verifying data structure consistency. In *Conference on Compiler Construction (Tool Demo)*.
- LEV-AMI, T. 2000. TVLA: A framework for Kleene based static analysis. M.S. thesis, Tel-Aviv University, Tel-Aviv, Israel.
- LEV-AMI, T., IMMERMAN, N., REPS, T., SAGIV, M., SRIVASTAVA, S., AND YORSH, G. 2005. Simulating reachability using first-order logic with applications to verification of linked data structures. Submitted for publication.
- LEV-AMI, T., REPS, T., SAGIV, M., AND WILHELM, R. 2000. Putting static analysis to work for verification: A case study. In *Proceedings of the International Symposium on Software Testing and Analysis*. 26–38.
- LEV-AMI, T. AND SAGIV, M. 2000. TVLA: A system for implementing static analyses. In *Proceedings of the Static Analysis Symposium*. 280–301.
- MCCUNE, W. 2001. Mace 2.0 reference manual and guide. <http://www-unix.mcs.anl.gov/AR/mace/>.
- MCMILLAN, K. L. 1999. Verification of infinite state systems by compositional model checking. In *Proceedings of the Correct Hardware Design and Verification Methods: 110th IFIP WG 10.5 Advanced Research Working Conference (CHARME)*. Lecture Notes in Computer Science, vol. 1703. Springer Verlag. 219–234.
- MEYER, A. R. 1975. Weak monadic second-order theory of successor is not elementary recursive. In *Logic Colloquium, Proceedings of the Symposium on Logic*. 132–154.

- MØLLER, A. AND SCHWARTZBACH, M. 2001. The pointer assertion logic engine. In *SIGPLAN Conference on Programming Language Design and Implementation*. 221–231.
- MORTIMER, M. 1975. On languages with two variables. *Zeitschrift für Math. Logik uematischend Grundlagen der Math 21*, 135–140.
- NIELSON, F., NIELSON, H., AND HANKIN, C. 1999. *Principles of Program Analysis*. Springer Verlag.
- NIELSON, F., NIELSON, H., AND SAGIV, M. 2000. A Kleene analysis of mobile ambients. In *Proceedings of the Programming Languages and Systems: 19th European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 1782. Springer Verlag. 305–319.
- RABIN, M. 1969. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc. 141*, 1–35.
- RAMALINGAM, G., VARSHAVSKY, A., FIELD, J., GOYAL, D., AND SAGIV, M. 2002. Deriving specialized program analyses for certifying component-client conformance. In *Proceedings of the PLDI Conference*. 83–94.
- REPS, T., SAGIV, M., AND YORSH, G. 2004. Symbolic implementation of the best transformer. In *Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, vol. 2937. Springer Verlag. 252–266.
- SAGIV, M., REPS, T., AND WILHELM, R. 1998. Solving shape-analysis problems in languages with destructive updating. *Trans. Program. Lang. Syst. 20*, 1 (Jan.), 1–50.
- SAGIV, M., REPS, T., AND WILHELM, R. 1999. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Language*. 105–118.
- SAGIV, M., REPS, T., AND WILHELM, R. 2002. Parametric shape analysis via 3-valued logic. *Trans. Program. Lang. Syst.*
- SHAHAM, R., YAHAV, E., KOLODNER, E., AND SAGIV, M. 2003. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proceedings of the Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 2694. Springer Verlag, 483–503.
- SHAPIRO, M. AND HORWITZ, S. 1997. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*. 1–14.
- STEENSGAARD, B. 1996. Points-to analysis in almost-linear time. In *Proceedings of the Symposium on Principles of Programming Languages*. 32–41.
- SU, Z., FÄHNDRICH, M., AND AIKEN, A. 2000. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the Symposium on Principles of Programming Languages*, T. Reps, Ed. 81–95.
- VAN FRAASSEN, B. 1966. Singular terms, truth-value gaps, and free logic. *J. Phil* 63, 17, 481–495.
- WEIDENBACH, C. 2006. SPASS: An automated theorem prover for first-order logic with equality. <http://spass.mpi-sb.mpg.de/index.html>.
- YAHAV, E. 2001. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the Symposium on Principles of Programming Languages 36*, 3, 27–40.
- YAHAV, E. AND RAMALINGAM, G. 2004. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 25–34.
- YAHAV, E. AND SAGIV, M. 2003. Automatically verifying concurrent queue algorithms. In *Electronic Notes in Theoretical Computer Science*, vol. 89. B. Cook et al. Eds. Elsevier.
- YORSH, G. 2003. Logical characterizations of heap abstractions. M.S. thesis, Tel-Aviv University, Tel-Aviv, Israel. <http://www.math.tau.ac.il/~gretay>.
- YORSH, G., REPS, T. W., AND SAGIV, M. 2004. Symbolically computing most precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 2988. Springer Verlag. 530–545.

Received April 2004; revised March 2005; accepted April 2005