Finite Differencing of Logical Formulas for Static Analysis

THOMAS REPS
University of Wisconsin and GrammaTech, Inc.
MOOLY SAGIV
Tel Aviv University
and
ALEXEY LOGINOV
GrammaTech, Inc.

This article concerns mechanisms for maintaining the value of an instrumentation relation (also known as a *derived relation* or *view*), defined via a logical formula over core relations, in response to changes in the values of the core relations. It presents an algorithm for transforming the instrumentation relation's defining formula into a *relation-maintenance formula* that captures what the instrumentation relation's new value should be. The algorithm runs in time linear in the size of the defining formula.

The technique applies to program analysis problems in which the semantics of statements is expressed using logical formulas that describe changes to core relation values. It provides a way to obtain values of the instrumentation relations that reflect the changes in core relation values produced by executing a given statement.

We present experimental evidence that our technique is an effective one: for a variety of benchmarks, the relation-maintenance formulas produced automatically using our approach yield the same precision as the best available hand-crafted ones.

At the time the research reported in the article was carried out, A. Loginov was affiliated with the University of Wisconsin.

Portions of the work appeared in *the 12th European Symposium on Programming* [Reps et al. 2003], R. Wilhelm's 60th birthday Festschrift [Loginov et al. 2007], and A. Loginov's Ph.D. dissertation [Loginov 2006].

The work was supported in part by NSF under grants CCR-{9619219, 9986308}, and CCF-{0540955, 0524051}, by the U.S.-Israel BSF under grant 96-00337, by ONR under contracts N00014-01-1-{0708,0796}, and by the von Humboldt and Guggenheim Foundations.

Authors' addresses: T. Reps, Computer Science Department, University of Wisconsin, and GrammaTech, Inc.; email: reps@cs.wisc.edu; M. Sagiv, School of Computer Science, Tel Aviv University, Israel; email: msagiv@post.tau.ac.il; A. Loginov, GrammaTech, Inc.; email: alexey@grammatech.com.

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 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.
© 2010 ACM 0164-0925/2010/08-ART24 $10.00

DOI 10.1145/1749608.1749613 http://doi.acm.org/10.1145/1749608.1749613

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures; dynamic storage management*; E.1 [**Data**]: Data Structures—*Graphs and networks; lists, stacks, and queues; records; trees*; E.2 [**Data**]: Data Storage Representations—*Composite structures; linked representations*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions; invariants; mechanical verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms: Algorithms, Languages, Theory, Verification

Additional Key Words and Phrases: Abstract interpretation, finite differencing, materialized view, shape analysis, static analysis, 3-valued logic

ACM Reference Format:
Reps, T., Sagiv, M., and Loginov, A. 2010. Finite differencing of logical formulas. ACM Trans. Program. Lang. Syst. 32, 6, Article 24 (August 2010), 55 pages.
DOI = 10.1145/1749608.1749613 http://doi.acm.org/10.1145/1749608.1749613

1. INTRODUCTION

This article addresses an instance of the following fundamental challenge in abstract interpretation:

> Given the concrete semantics for a language and a desired abstraction, how does one create the associated abstract transformers?

The problem that we address arises in program analysis problems in which the semantics of statements is expressed using logical formulas that describe changes to core relation values. When instrumentation relations (defined via logical formulas over the core relations) have been introduced to refine an abstraction, the challenge is to develop a method for obtaining values of the instrumentation relations that reflect the changes in core relation values [Graf and Saïdi 1997; Das et al. 1999; McMillan 1999; Sagiv et al. 2002; Ball et al. 2001]. The algorithm presented in this article provides a way to create formulas that maintain correct values for the instrumentation relations, and thereby provides a way to generate, completely automatically, the part of the transformers of an abstract semantics that deals with instrumentation relations. The algorithm runs in time linear in the size of the instrumentation relation's defining formula.

This research was motivated by our work on static analysis based on 3-valued logic [Sagiv et al. 2002]; however, any analysis method that relies on logic—2-valued or 3-valued—to express a program's semantics may be able to benefit from these techniques.

In our setting, two related logics come into play: an ordinary 2-valued logic, as well as a related 3-valued logic. A memory configuration, or store, is modeled by what logicians call a *logical structure*; an individual of the structure's universe either models a single memory element or, in the case of a *summary individual*, it models a collection of memory elements. A run of the analyzer carries out an abstract interpretation to collect a set of structures at each program point P. This involves finding the least fixed point of a certain set of equations. When the fixed point is reached, the structures that have been collected at program point P describe a superset of all the execution states that can occur at P. To determine whether a property always holds at P, one checks 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. The TVLA system (*T*hree-*V*alued-*L*ogic *A*nalyzer) implements this approach [Lev-Ami and Sagiv 2000; TVLA].

Summary individuals play a crucial role. They are used to ensure that abstract descriptors have an a priori bounded size, which guarantees that a fixed point is always reached. However, the constraint of working with limited-size descriptors implies a loss of information about the store. Intuitively, certain properties of concrete individuals are lost due to abstraction, which groups together multiple individuals into summary individuals: a property can be true for some concrete individuals of the group but false for other individuals. It is for this reason that 3-valued logic is used; uncertainty about a property's value is captured by means of the third truth value, 1/2.

An advantage of using 2- and 3-valued logic as the basis for static analysis is that the language used for extracting information from the concrete world and the abstract world is identical: every syntactic expression (i.e., every logical formula) can be interpreted either in the 2-valued world or the 3-valued world. The consistency of the 2-valued and 3-valued viewpoints is ensured by a basic theorem that relates the two logics [Sagiv et al. 2002, Theorem 4.9]. This provides a partial answer to the fundamental challenge posed before: formulas that define the concrete semantics, when interpreted in 2-valued logic, define a sound abstract semantics when interpreted in 3-valued logic [Sagiv et al. 2002].

Unfortunately, unless some care is taken in the design of an analysis, there is a danger that as abstract interpretation proceeds, the indefinite value 1/2 will become pervasive. This can destroy the ability to recover interesting information from the 3-valued structures collected (although soundness is maintained). A key role in combating indefiniteness is played by *instrumentation relations*, which record auxiliary information in a logical structure. The benefit of introducing instrumentation relations was annunciated as the Instrumentation Principle.

Observation 1.1 (*Instrumentation Principle* [*Sagiv et al. 2002, Observation* 2.8]). Suppose that $S^{\#}$ is a 3-valued structure that represents the 2-valued structure S. By explicitly "storing" in $S^{\#}$ the values that a formula ψ has in S, it is sometimes possible to extract more precise information from $S^{\#}$ than can be obtained just by evaluating ψ in $S^{\#}$.

Instrumentation relations provide a mechanism to fine-tune an abstraction: an instrumentation relation, which is defined by a logical formula ψ over the core relation symbols, captures a property that may or may not be possessed by a structure, an individual memory cell, or a tuple of memory cells (according to whether ψ is a nullary, unary, or k-ary formula, respectively). For instance, the following formulas define nullary, unary, and binary instrumentation relations relating to cycles (of length one or more) along n edges, where the * operator denotes transitive closure.

Nullary (Does the structure contain a cycle?): $\psi_{c_0}() \overset{\text{def}}{=} \exists v_1, v_2 : n(v_1, v_2) \wedge n^*(v_2, v_1)$
Unary (Is v_1 on a cycle?): $\psi_{c_1}(v_1) \overset{\text{def}}{=} \exists v_2 : n(v_1, v_2) \wedge n^*(v_2, v_1)$
Binary (Is n edge $v_1 \rightarrow v_2$ part of a cycle?): $\psi_{c_2}(v_1, v_2) \overset{\text{def}}{=} n(v_1, v_2) \wedge n^*(v_2, v_1)$

In general, the introduction of additional instrumentation relations refines an abstraction into one that is prepared to track finer distinctions among stores. For reasons discussed in Section 3, the values of instrumentation relations are stored and maintained in response to the store transformations performed by program statements. In many cases, this technique allows more precise properties of the program's stores to be established.

Problem Statement and Contributions. From the standpoint of the concrete semantics, instrumentation relations represent cached information that could always be recomputed by reevaluating the instrumentation relation's defining formula in the local state. From the standpoint of the abstract semantics, however, reevaluating a formula in the local (3-valued) state can lead to a drastic loss of precision. To gain maximum benefit from instrumentation relations, an abstract-interpretation algorithm must obtain their values in some other way. We call this problem the *instrumentation-relation-maintenance problem* (often shortened to the "relation-maintenance problem"). To summarize, the problem that we address is the following:

> Given a formula ψ_p that defines an instrumentation relation p, together with formulas τ_c that specify how each core relation c is transformed by transformer τ, create a relation-maintenance formula for p.

To reduce the loss of precision, the solution to the relation-maintenance problem developed in this article uses an incremental computation strategy. After a transition via transformer τ from abstract state σ to abstract state σ', the new value that instrumentation relation p should have is computed from the stored value of p in σ.

The contributions of the work reported in the article can be summarized as follows.

- We give an algorithm for solving the relation-maintenance problem. The algorithm works by applying a finite-differencing transformation to p's defining formula ψ_p. The algorithm runs in time linear in the size of ψ_p.
- We present experimental evidence that our technique is an effective one, at least for the analysis of programs that manipulate (cyclic and acyclic) singly-linked lists, doubly-linked lists, and binary trees, and for certain sorting programs. In particular, the relation-maintenance formulas produced automatically using our approach are as effective for maintaining precision as the best available hand-crafted ones.

Organization. The remainder of the article is organized as follows: Section 2 introduces terminology and notation. Section 3 defines the relation-maintenance problem. Section 4 provides intuition behind our solution, which is presented in Section 5 and Section 6. Section 5 presents a method for generating maintenance formulas for instrumentation relations. Section 6 discusses extensions to handle instrumentation relations that use transitive closure. Section 7 presents experimental results. Section 8 discusses related work. Section 9 presents some concluding remarks. Finally, the electronic Appendix (available in the ACM Digital Library) presents a proof of the correctness of our solution to the relation-maintenance problem.

2. BACKGROUND

This section introduces terminology and notation; it presents the logic that we employ and describes the use of logical structures for representing memory stores.

The first half of Section 2.1 introduces 2-valued first-order logic with transitive closure. These concepts are standard in logic. The second half of Section 2.1 presents a straightforward extension of the logic to the 3-valued setting, in which a third truth value—1/2—is introduced to denote uncertainty. Section 2.2 summarizes the program analysis framework described in Sagiv et al. [2002]. In that approach, memory configurations are encoded as 2-valued logical structures. The semantics of programs, as well as properties of memory configurations, are encoded using formulas. Abstract interpretation [Cousot and Cousot 1977] is performed to compute, at each point in the program being analyzed, a set of 3-valued logical structures that overapproximates the memory configurations that can arise at that point.

2.1 First-Order Logic with Transitive Closure

2-*Valued first-order logic with transitive closure*. The syntax of first-order formulas with equality and reflexive transitive closure is defined as follows.

Definition 2.1. Let R_i denote a set of arity-i relation symbols,[1] with $eq \in R_2$. A *formula* over the *vocabulary* $\mathcal{R} = \bigcup_i R_i$ is defined by

$$\begin{array}{lll} p \in \mathcal{R}_k & \varphi ::= & \mathbf{0} \mid \mathbf{1} \mid p(v_1, \ldots, v_k) \\ \varphi \in \mathit{Formulas} & & \mid (\neg\varphi_1) \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 \vee \varphi_2) \mid (\exists v\colon \varphi_1) \mid (\forall v\colon \varphi_1) \\ v \in \mathit{Variables} & & \mid (\mathbf{RTC}\ v_1', v_2'\colon \varphi_1)(v_1, v_2). \end{array}$$

A formula of the form $\mathbf{0}$, $\mathbf{1}$, or $p(v_1, \ldots, v_k)$ is called an *atomic formula*.

The set of *free variables* of a formula is defined as usual. "**RTC**" stands for reflexive transitive closure. In $\varphi \equiv (\mathbf{RTC}\ v_1', v_2'\colon \varphi_1)(v_1, v_2)$, if φ_1's free-variable set is V, we require $v_1, v_2 \notin V$. The free variables of φ are $(V - \{v_1', v_2'\}) \cup \{v_1, v_2\}$.

We use several shorthand notations: $(v_1 = v_2) \stackrel{\text{def}}{=} eq(v_1, v_2)$; $(v_1 \neq v_2) \stackrel{\text{def}}{=} \neg eq(v_1, v_2)$; and for a binary relation p, $p^*(v_1, v_2) \stackrel{\text{def}}{=} (\mathbf{RTC}\ v_1', v_2' : p(v_1', v_2'))(v_1, v_2)$. We also use a C-like syntax for conditional expressions: $\varphi_1 \mathbin{?} \varphi_2 : \varphi_3$.[2] The order

[1] Instead of introducing function symbols, we encode a function of arity i by means of a relation of arity $i + 1$, together with logical constraints (described in Section 2.2.2 in the discussion of Figure 10).

[2] In 2-valued logic, one can think of $\varphi_1 \mathbin{?} \varphi_2 : \varphi_3$ as a shorthand for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3)$. In 3-valued logic, it becomes a shorthand for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$ [Reps et al. 2002].

of precedence among the connectives, from highest to lowest, is as follows: $\neg$, $\wedge$, $\vee$, $\forall$, and $\exists$. We drop parentheses wherever possible, except for emphasis.

Definition 2.2. A *2-valued interpretation* over $\mathcal{R}$ 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 relation symbol $p \in \mathcal{R}_k$ to a truth-valued function: $\iota^S(p)\colon (U^S)^k \to \{0, 1\}$. In addition, (i) for all $u \in U^S$, $\iota^S(eq)(u, u) = 1$, and (ii) for all $u_1, u_2 \in U^S$ such that u_1 and u_2 are distinct individuals, $\iota^S(eq)(u_1, u_2) = 0$.

An *assignment* Z is a function that maps variables to individuals (i.e., it has the functionality $Z\colon \{v_1, v_2, \ldots\} \to U^S$). When Z is defined on all free variables of a formula φ, we say that Z is *complete* for φ. (We generally assume that every assignment 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 $\{0, 1\}$; it is defined inductively as follows.

$$\begin{array}{rlrl}
[\![\mathbf{0}]\!]_2^S(Z) = 0 & & [\![\varphi_1 \wedge \varphi_2]\!]_2^S(Z) = \min([\![\varphi_1]\!]_2^S(Z), [\![\varphi_2]\!]_2^S(Z)) \\
[\![\mathbf{1}]\!]_2^S(Z) = 1 & & [\![\varphi_1 \vee \varphi_2]\!]_2^S(Z) = \max([\![\varphi_1]\!]_2^S(Z), [\![\varphi_2]\!]_2^S(Z)) \\
[\![p(v_1, \ldots, v_k)]\!]_2^S(Z) = \iota^S(p)(Z(v_1), \ldots, Z(v_k)) & & [\![\exists v\colon \varphi_1]\!]_2^S(Z) = \max_{u \in U^S} [\![\varphi_1]\!]_2^S(Z[v \mapsto u]) \\
[\![\neg\varphi_1]\!]_2^S(Z) = 1 - [\![\varphi_1]\!]_2^S(Z) & & [\![\forall v\colon \varphi_1]\!]_2^S(Z) = \min_{u \in U^S} [\![\varphi_1]\!]_2^S(Z[v \mapsto u])
\end{array}$$

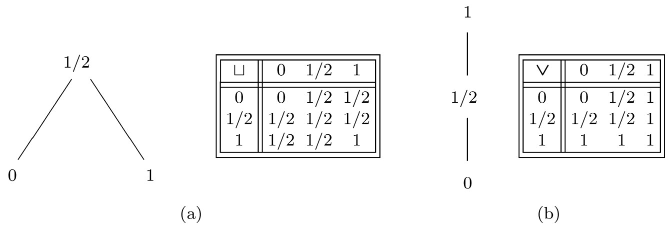
$$[\![(\mathbf{RTC}\ v_1', v_2'\colon \varphi_1)(v_1, v_2)]\!]_2^S(Z) = \begin{cases} 1 & \text{if } Z(v_1) = Z(v_2) \\ \displaystyle\max_{\substack{n \geq 1, \\ u_1, \ldots, u_{n+1} \in U, \\ Z(v_1) = u_1, \\ Z(v_2) = u_{n+1}}} \min_{i=1}^{n} [\![\varphi_1]\!]_2^S(Z[v_1' \mapsto u_i, v_2' \mapsto u_{i+1}]) & \text{otherwise} \end{cases}$$

S and Z satisfy φ if $[\![\varphi]\!]_2^S(Z) = 1$. The set of 2-valued structures is denoted by $\mathcal{S}_2[\mathcal{R}]$, where "$[\mathcal{R}]$" is dropped if the vocabulary $\mathcal{R}$ is understood.

3-*Valued logic and embedding*. In 3-valued logic, the formulas that we work with are identical to the ones used in 2-valued logic. At the semantic level, a third truth value—1/2—is introduced to denote uncertainty.

Definition 2.3. The truth values 0 and 1 are *definite values*; 1/2 is an *indefinite value*. For $l_1, l_2 \in \{0, 1/2, 1\}$, the *information order* is defined as follows: $l_1 \sqsubseteq l_2$ iff $l_1 = l_2$ or $l_2 = 1/2$. $l_1 \sqsubseteq l_2$ denotes that l_1 is at least as definite as l_2. We use $l_1 \sqsubset l_2$ when $l_1 \sqsubseteq l_2$ and $l_1 \neq l_2$. The symbol $\sqcup$ denotes the least-upper-bound operation with respect to $\sqsubseteq$.

As shown in Figure 1, we place two orderings on 0, 1, and 1/2: (i) the *information order*, denoted by $\sqsubseteq$ and illustrated in Figure 1(a), captures "(un)certainty"; (ii) the *logical order*, shown in Figure 1(b), defines the meaning of $\wedge$ and $\vee$; that is, $\wedge$ and $\vee$ are meet and join in the logical order. 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). Because $\varphi_1 \,?\, \varphi_2 : \varphi_3$ is treated as a shorthand for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$



$\sqcup$	0	1/2	1
0	0	1/2	1/2
1/2	1/2	1/2	1/2
1	1/2	1/2	1

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

Fig. 1. (a) The information order ($\sqsubseteq$) and its join operation ($\sqcup$). (b) The logical order and its join operation ($\vee$).

in 3-valued logic [Reps et al. 2002], the value of $1/2 \,?\, V_1 : V_2$ equals $V_1 \sqcup V_2$. We now generalize Definition 2.2 to define the meaning of a formula with respect to a 3-valued structure.

Definition 2.4. A *3-valued interpretation* over $\mathcal{R}$ 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 relation symbol $p \in \mathcal{R}_k$ to a truth-valued function: $\iota^S(p)\colon (U^S)^k \to \{0, 1/2, 1\}$. 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 individuals, $\iota^S(eq)(u_1, u_2) = 0$.

For an assignment Z, the *(3-valued) meaning* of a formula φ, denoted by $[\![\varphi]\!]_3^S(Z)$, yields a truth value in $\{0, 1/2, 1\}$. The meaning of φ is defined exactly as in Definition 2.2, but interpreted over $\{0, 1/2, 1\}$. S and Z *potentially satisfy* φ if $[\![\varphi]\!]_3^S(Z) \sqsupseteq 1$. The set of 3-valued structures is denoted by $\mathcal{S}_3[\mathcal{R}]$, where "$[\mathcal{R}]$" is dropped if the vocabulary $\mathcal{R}$ is understood.

Definition 2.4 requires that for each individual u, the value of $\iota^S(eq)(u,u)$ is 1 or 1/2. An individual for which $\iota^S(eq)(u,u) = 1/2$ is called a *summary individual*. In the program analysis framework of Sagiv et al. [2002], a summary individual abstracts one or more nodes of a data structure, and hence can represent more than one concrete memory cell.

The embedding ordering on structures is defined as follows.

Definition 2.5. 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 for every relation symbol $p \in \mathcal{R}_k$ and for all $u_1, \ldots, u_k \in U^S$, $\iota^S(p)(u_1, \ldots, u_k) \sqsubseteq \iota^{S'}(p)(f(u_1), \ldots, f(u_k))$. We say that *S can be embedded in* S' (denoted by $S \sqsubseteq S'$) if there exists a function f such that $S \sqsubseteq^f S'$.

The embedding theorem says that 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 $(f \circ Z)(v) = f(Z(v))$.

THEOREM 2.6. (EMBEDDING THEOREM [SAGIV ET AL. 2002, 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, $[\![\varphi]\!]_3^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S'}(f \circ Z)$.

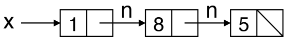


Fig. 2. A possible store for a linked list.

```
typedef struct node {
  struct node *n;
  int data;
} *List;
```

(a)

Relation	Intended Meaning
$eq(v_1, v_2)$	Do v_1 and v_2 denote the same memory cell?
$x(v)$	Does pointer variable `x` point to memory cell v?
$n(v_1, v_2)$	Does the `n` field of v_1 point to v_2?
$dle(v_1, v_2)$	Is the `data` field of v_1 less than or equal to that of v_2?

(b)

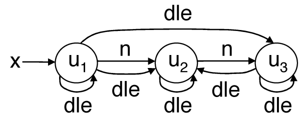
Fig. 3. (a) Declaration of a linked-list datatype in C. (b) Core relations used for representing the stores manipulated by programs that use type `List`.

In the rest of the article, we will denote 2-valued structures by S (possibly with subscripts and primes) and 3-valued structures by $S^{\#}$ (possibly with subscripts).

2.2 Stores as Logical Structures and their Abstractions

Program analysis via 3-valued logic. The remainder of this section summarizes the program analysis framework described in Sagiv et al. [2002]. In that approach, concrete memory configurations (i.e., *stores*) are encoded as logical structures (associated with a *vocabulary* of relation symbols with given arities) in terms of a fixed collection of *core relations*, $\mathcal{C}$. Core relations are part of the underlying semantics of the language to be analyzed; they record atomic properties of stores. For instance, Figure 3 gives the definition of a C linked-list datatype, and lists the relations that would be used to represent the stores manipulated by programs that use type `List`, such as the store in Figure 2. (The core relations are fixed for a given combination of language and datatype; in general, different languages and datatypes require different collections of core relations.) 2-valued logical structures then represent memory configurations: the individuals of the structure are the set of memory cells; a nullary relation represents a Boolean variable of the program; a unary relation represents either a pointer variable or a Boolean-valued field of a record; and a binary relation represents a pointer field of a record. In Figure 3, unary relations represent pointer variables, and binary relation n represents the `n`-field of a `List` cell. Numeric-valued variables and numeric-valued fields (such as `data`) can be modeled by introducing other relations, such as the binary relation *dle* (which stands for "`data` less-than-or-equal-to") listed in Figure 3; *dle* captures the relative order of two nodes' `data` values. (Alternatively, numeric-valued entities can be handled by combining abstractions of logical structures with previously known techniques for creating numeric abstractions [Gopan et al. 2004].) Figure 4 shows 2-valued structure S_4, which represents the store of Figure 2 using the relations of Figure 3. S_4 has three individuals, u_1, u_2, and u_3, which represent the three list elements.

Information about a concrete memory configuration encoded as a logical structure can be extracted from the logical structure by evaluating formulas.



	x
u_1	1
u_2	0
u_3	0

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

dle	u_1	u_2	u_3
u_1	1	1	1
u_2	0	1	0
u_3	0	1	1

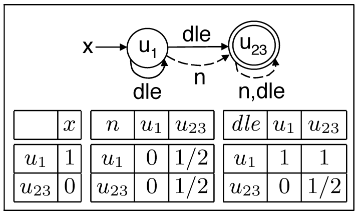
Fig. 4. A logical structure S_4 that represents the store shown in Figure 2 in graphical and tabular forms using the relations of Figure 3 (Relation *eq* is not shown explicitly; each node has an *eq* self-loop, and the relation in tabular form is the identity matrix.)

A concrete operational semantics is defined by specifying a *structure transformer* $st_{(n_1,n_2)}$ for each outgoing Control-Flow Graph (CFG) edge (n_1, n_2). (Ordinarily (n_1, n_2) is understood, and we just write *st*.) A structure transformer is specified by providing a collection of *relation-transfer formulas*, $\tau_{c,st}$, one for each core relation c. These formulas define how the core relations of a 2-valued logical structure S_1 that arises at n_1 are transformed by $st_{(n_1,n_2)}$ to create a 2-valued logical structure S_2 at n_2; typically, they define the value of relation c in S_2 as a function of c's value in S_1 and the values of other core relations in S_1. For instance, Figure 9, described in more detail later in this section, shows that the value of unary relation y in a structure transformed by the structure transformer corresponding to the statement $\mathrm{y} = \mathrm{x}$ is defined as a function of the value of unary relation x, namely: $\tau_{y,\mathrm{y}=\mathrm{x}}(v) = x(v)$. We use the notation $[\![st]\!]_2(S_1)$ to denote the transformation of S_1 by structure transformer *st*.

Transformer *st* may optionally have a *precondition formula*, which filters out structures that should not follow the transition along (n_1, n_2). The postcondition operator*post* for edge (n_1, n_2) is defined by lifting (n_1, n_2)'s structure transformer to sets of structures.

Abstract stores are 3-valued logical structures. Concrete stores are abstracted to abstract stores by means of *embedding functions*—onto functions that map individuals of a 2-valued structure S to those of a 3-valued structure $S^{\#}$. The embedding theorem ensures that every piece of information extracted from $S^{\#}$ by evaluating a formula φ is a conservative approximation ($\sqsupseteq$) of the information extracted from S by evaluating φ.

To obtain a computable abstract domain, we ensure that the size of the 3-valued structures used to represent memory configurations is always bounded. We do this by defining an equivalence relation on individuals and considering the (bounded-size) quotient structure with respect to this equivalence relation; in particular, each individual of a 2-valued logical structure (representing a concrete memory cell) is mapped to an individual of a 3-valued logical structure according to the vector of values that the concrete individual has for a user-chosen collection of unary abstraction relations. Intuitively, this equivalence relation maps a group of individuals, which are indistinguishable according to the set of (unary) abstraction relations $\mathcal{A}$, to a single individual:



	x
u_1	1
u_{23}	0

n	u_1	u_{23}
u_1	0	1/2
u_{23}	0	1/2

dle	u_1	u_{23}
u_1	1	1
u_{23}	0	1/2

Fig. 5. A 3-valued structure $S_5^\#$ that is the canonical abstraction of structure S_4.

Definition (*Canonical Abstraction*). Let $S \in \mathcal{S}_2[\mathcal{R}]$, and let $\mathcal{A} \subseteq \mathcal{R}_1$ be some chosen (nonempty) subset of the unary relation symbols. The relations in $\mathcal{A}$ are called *abstraction relations*; they define the following equivalence relation $\simeq_\mathcal{A}$ on U^S.

$$u_1 \simeq_\mathcal{A} u_2 \iff \text{for all } p \in \mathcal{A}, \iota^S(p)(u_1) = \iota^S(p)(u_2)$$

Additionally, *abstraction relations* define the surjective function $f_\mathcal{A} : U^S \to U^S/\simeq_\mathcal{A}$, such that $f_\mathcal{A}(u) = [u]_{\simeq_\mathcal{A}}$, which maps an individual to its equivalence class. The *canonical abstraction* of S with respect to $\mathcal{A}$ (denoted by $f_\mathcal{A}(S)$) performs the join (in the information order) of relation values, thereby introducing 1/2's: for every $p \in \mathcal{R}_k$,

$$\iota^{f_\mathcal{A}(S)}(p)(u'_1, \ldots, u'_k) = \bigsqcup_{\substack{(u_1,\ldots,u_k) \in (U^S)^k, \text{ such that} \\ f_\mathcal{A}(u_i) = u'_i \in U^S/\simeq_\mathcal{A}, 1 \le i \le k}} \iota^S(p)(u_1, \ldots, u_k). \qquad (1)$$

If $\mathcal{A} = \{x\}$, the canonical abstraction of 2-valued logical structure S_4 is $S_5^\#$, shown in Figure 5, with $f_\mathcal{A}(u_1) = u_1$ and $f_\mathcal{A}(u_2) = f_\mathcal{A}(u_3) = u_{23}$. In addition to S_4, $S_5^\#$ represents any list with two or more elements that is pointed to by program variable x, and in which the first element's data value is (definitely) less than the data values in the rest of the list (note the absence of either a 1-valued or 1/2-valued *dle* edge from individual u_{23} to individual u_1). The following graphical notation is used for depicting 3-valued logical structures.

—Individuals are represented by circles containing their names and (non-0) values for unary relations. Summary individuals are represented by double circles.

—A unary relation p corresponding to a pointer-valued program variable is represented by a solid arrow from p to the individual u for which $p(u) = 1$, and by the absence of a p-arrow to each node u' for which $p(u') = 0$. (If $p = 0$ for all individuals, the relation name p is not shown.)

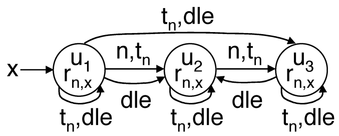
—A binary relation q is represented by a solid arrow labeled q between each pair of individuals u_i and u_j for which $q(u_i, u_j) = 1$, and by the absence of a q-arrow between pairs u'_i and u'_j for which $q(u'_i, u'_j) = 0$.

—Relations with value 1/2 are represented by dashed arrows.

Canonical abstraction ensures that each 3-valued structure is no larger than some fixed size, known a priori. While canonical abstraction is defined on 2-valued structures, its operations can be applied to 3-valued structures as well,

p	**Intended Meaning**	ψ_p
$is_n(v)$	Do n fields of two or more list nodes point to v?	$\exists\, v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$
$t_n(v_1, v_2)$	Is v_2 reachable from v_1 along zero or more n fields?	$n^*(v_1, v_2)$
$r_{n,x}(v)$	Is v reachable from pointer variable x along zero or more n fields?	$\exists\, v_1 : x(v_1) \wedge t_n(v_1, v)$
$c_n(v)$	Is v on a directed cycle of n fields?	$\exists\, v_1 : n(v_1, v) \wedge t_n(v, v_1)$

Fig. 6. Defining formulas of some commonly used instrumentation relations. The relation name is_n abbreviates "is-shared". There is a separate reachability relation $r_{n,x}$ for every program variable x. (Recall that $v_1 \neq v_2$ is a shorthand for $\neg eq(v_1, v_2)$, and $n^*(v_1, v_2)$ is a shorthand for $(\mathbf{RTC}\ v_1', v_2' : n(v_1', v_2'))(v_1, v_2)$.)



	x	$r_{n,x}$	c_n
u_1	1	1	0
u_2	0	1	0
u_3	0	1	0

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

t_n	u_1	u_2	u_3
u_1	1	1	1
u_2	0	1	1
u_3	0	0	1

dle	u_1	u_2	u_3
u_1	1	1	1
u_2	0	1	0
u_3	0	1	1

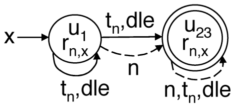
Fig. 7. A logical structure S_7, which represents the store shown in Figure 2, in graphical and tabular forms using the relations of Figures 3 and 6.

possibly producing more abstract structures (i.e., ones with fewer individuals). In a slight abuse of terminology, we will sometimes discuss the application of canonical abstraction to 3-valued structures.

2.2.1 *Instrumentation Relations.* The abstraction function on which an analysis is based, and hence the precision of the analysis defined, can be tuned by (i) choosing to equip structures with additional *instrumentation relations* to record derived properties, and (ii) varying which of the unary core and unary instrumentation relations are used as the set of abstraction relations. The set of instrumentation relations is denoted by $\mathcal{I}$. Each arity-k relation symbol $p \in \mathcal{I}$ is defined by an *instrumentation-relation definition formula* $\psi_p(v_1, \ldots, v_k)$. Instrumentation relations may appear in the defining formulas of other instrumentation relations as long as there are no circular dependences.

The introduction of unary instrumentation relations that are used as abstraction relations provides a way to control which concrete individuals are merged together into an abstract individual, and thereby control the amount of information lost by abstraction. Instrumentation relations that involve reachability properties, which can be defined using **RTC**, often play a crucial role in the definitions of abstractions. For instance, in program analysis applications, reachability properties from specific pointer variables have the effect of keeping disjoint sublists summarized separately. Figure 6 lists some instrumentation relations that are important for the analysis of programs that use type `List`.

Figure 7 shows 2-valued structure S_7, which represents the store of Figure 2 using the core relations of Figure 3, as well as the instrumentation relations



	x	$r_{n,x}$	c_n
u_1	1	1	0
u_{23}	0	1	0

n	u_1	u_{23}
u_1	0	1/2
u_{23}	0	1/2

t_n	u_1	u_{23}
u_1	1	1
u_{23}	0	1/2

dle	u_1	u_{23}
u_1	1	1
u_{23}	0	1/2

Fig. 8. A 3-valued structure $S^{\#}_8$ that is the canonical abstraction of structure S_7.

of Figure 6. If all unary relations are abstraction relations ($\mathcal{A} = \mathcal{R}_1$), the canonical abstraction of 2-valued logical structure S_7 is $S^{\#}_8$, shown in Figure 8, with $f_{\mathcal{A}}(u_1) = u_1$ and $f_{\mathcal{A}}(u_2) = f_{\mathcal{A}}(u_3) = u_{23}$.

2.2.2 *Abstract Interpretation.* For each kind of statement in the programming language, the abstract semantics is again defined by a collection of formulas: the same relation-transfer formula that defines the concrete semantics, in the case of a core relation, and, in the case of an instrumentation relation p, by a *relation-maintenance formula* $\mu_{p,st}$.[3]

In our context, abstract interpretation collects a set of 3-valued structures at each program point. It can be implemented as an iterative procedure that finds the least fixed point of a certain set of equations [Sagiv et al. 2002]. (It is important to understand that although the analysis framework is based on logic, it is model-theoretic, not proof-theoretic: the abstract interpretation collects sets of 3-valued logical structures—i.e., abstracted models; its actions do not rely on deduction or theorem proving.) When the fixed point is reached, the structures that have been collected at program point P describe a superset of all the execution states that can occur at P. To determine whether a property always holds at P, one checks whether it holds in all of the structures that were collected there.

Figure 9 illustrates the abstract execution of the statement $\mathrm{y} = \mathrm{x}$ on a 3-valued logical structure that represents concrete lists of length 2 or more. Instrumentation relations and relation-maintenance formulas have been omitted from the figure. The abstract execution of the statement $\mathrm{y} = \mathrm{x}$ is revisited in Example 3.2 of Section 3, which discusses relation-maintenance formulas.

Other operations on logical structures. *focus*[φ] is a heuristic that elaborates a 3-valued structure, causing it to be replaced by a collection of more precise structures that, taken together, represent the same set of concrete stores;[4] the criterion for refinement is to ensure that the formula φ evaluates to a definite value for all assignments to φ's free variables. The operation thus brings φ "into focus."

[3] In Sagiv et al. [2002], relation-transfer formulas and relation-maintenance formulas are both called "relation-update formulas". Here we use separate terms so that we can refer easily to relation-maintenance formulas, which are the main subject of this article. The term "relation-maintenance formula" emphasizes the connection to work in the database community on *view maintenance* (see Section 8). ("View updating" is something different: an update is made to the value of a view relation and changes are propagated back to the base relations.)

[4] This operation can be viewed as a partial concretization.

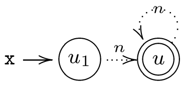
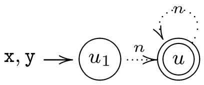
<table>
<tr><td>Structure before</td><td><b>unary rels.</b><table><tr><th>indiv.</th><th>x</th><th>y</th></tr><tr><td>u_1</td><td>1</td><td>0</td></tr><tr><td>u</td><td>0</td><td>0</td></tr></table></td><td><b>binary rels.</b><table><tr><th>n</th><th>u_1</th><th>u</th></tr><tr><td>u_1</td><td>0</td><td>1/2</td></tr><tr><td>u</td><td>0</td><td>1/2</td></tr></table></td><td><table><tr><th>eq</th><th>u_1</th><th>u</th></tr><tr><td>u_1</td><td>1</td><td>0</td></tr><tr><td>u</td><td>0</td><td>1/2</td></tr></table></td><td> </td></tr>
<tr><td>Statement</td><td colspan="4"><code>y = x</code></td></tr>
<tr><td>Relation-transfer formulas</td><td colspan="4">$\tau_{x,\mathtt{y=x}}(v) = x(v)$
$\tau_{y,\mathtt{y=x}}(v) = x(v)$
$\tau_{n,\mathtt{y=x}}(v_1, v_2) = n(v_1, v_2)$
$\tau_{eq,\mathtt{y=x}}(v_1, v_2) = eq(v_1, v_2)$</td></tr>
<tr><td>Structure after</td><td><b>unary rels.</b><table><tr><th>indiv.</th><th>x</th><th>y</th></tr><tr><td>u_1</td><td>1</td><td>1</td></tr><tr><td>u</td><td>0</td><td>0</td></tr></table></td><td><b>binary rels.</b><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><table><tr><th>eq</th><th>u_1</th><th>u</th></tr><tr><td>u_1</td><td>1</td><td>0</td></tr><tr><td>u</td><td>0</td><td>1/2</td></tr></table></td><td> </td></tr>
</table>

Fig. 9. The relation-transfer formulas for x, y, and n express a transformation on logical structures that corresponds to the semantics of $\mathtt{y} = \mathtt{x}$.

By invoking *focus* before applying each structure transformer, focusing is used to reduce the number of indefinite values that arise when relation-transfer and relation-maintenance formulas are evaluated in 3-valued structures. The *focus* formulas aim to sharpen the values of relations when applied to the individuals that are affected by the transformer. (This often involves the *materialization* of a concrete individual out of a summary individual.) For program analysis applications, it was proposed in Sagiv et al. [2002] that for a statement of the form *lhs* = *rhs*, the focus formula should identify the memory cells that correspond to the *L*-value of *lhs* and the *R*-value of *rhs*. This ensures that the application of an abstract transformer performs a *strong update* of the values of core relations that represent pointer variables and fields that are updated by the statement, that is, does not set those values to 1/2.

Not all logical structures represent admissible stores. To exclude structures that do not, we impose integrity constraints. For instance, relation $x(v)$ of Figure 3 captures whether pointer variable x points to memory cell v; x would be given the attribute "unique," which imposes the integrity constraint that x can hold for at most one individual in any structure: $\forall\, v_1, v_2 : x(v_1) \wedge x(v_2) \Rightarrow v_1 = v_2$. This formula evaluates to 1 in any 2-valued logical structure that corresponds to an admissible store. Figure 10 gives the list of relation attributes that are used in this article, together with their intended meaning. The precise integrity constraints used to enforce the intended meaning of each attribute are introduced where the attribute is discussed.

Integrity constraints contribute to the concretization function (γ) for our abstraction [Yorsh et al. 2007]. Integrity constraints are enforced by *coerce*, a clean-up operation that may "sharpen" a 3-valued logical structure by setting an indefinite value (1/2) to a definite value (0 or 1), or discard a structure entirely if an integrity constraint is definitely violated by the structure (for example, if it cannot represent any admissible store). To help prevent an analysis from losing precision, *coerce* is applied at certain steps of the algorithm, for example, after the application of an abstract transformer.

In addition, most of the operations described in this section are not constrained to manipulate 3-valued structures that are images of canonical

Attribute	**Arity of Relation**	**Intended Meaning**
$unique(p)$	$p \in \mathcal{R}_1$	$p(v)$ holds for at most one assignment to v
$function(p)$	$p \in \mathcal{R}_2$	For each assignment to v_1, $p(v_1, v_2)$ holds for at most one assignment to v_2
$invfunction(p)$	$p \in \mathcal{R}_2$	For each assignment to v_2, $p(v_1, v_2)$ holds for at most one assignment to v_1
$acyclic(p)$	$p \in \mathcal{R}_2$	$p(v_1, v_2)$ defines an acyclic graph
$tree(p)$	$p \in \mathcal{R}_2$	$p(v_1, v_2)$ defines a tree-shaped graph

Fig. 10. The meaning of relation attributes used in this article.

abstraction; they rely on the embedding theorem, which applies to any pair of structures for which one can be embedded into the other. Thus, it is not necessary to perform canonical abstraction after the application of each abstract structure transformer. To ensure that abstract interpretation terminates, it is only necessary that canonical abstraction be applied somewhere in each loop, for instance, at the target of each backedge in the CFG.

3. THE PROBLEM: MAINTAINING INSTRUMENTATION RELATIONS

The execution of a statement *st* transforms a 3-valued structure $S_1^\#$, which represents a store that arises just before *st*, into a new structure $S_2^\#$ which represents the corresponding store just after *st* executes. The structure that consists of just the core relations of $S_2^\#$ is called a *proto-structure*, denoted by $S_{proto}^\#$. The creation of $S_{proto}^\#$ from $S_1^\#$, denoted by $S_{proto}^\# := [\![st]\!]_3(S_1^\#)$, can be expressed as

$$\begin{aligned}&\text{for each } c \in \mathcal{C} \text{ and } u_1, \ldots, u_k \in U^{S_1^\#},\\ &\quad \iota^{S_{proto}^\#}(c)(u_1, \ldots, u_k) := [\![\tau_{c,st}(v_1, \ldots, v_k)]\!]_3^{S_1^\#}([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]).\end{aligned} \qquad (2)$$

In general, if we compare the various relations of $S_{proto}^\#$ with those of $S_1^\#$, some tuples will have been added and others will have been deleted.

We now come to the crux of the matter: Suppose that ψ_p defines instrumentation relation p; how should the static analysis engine obtain the value of p in $S_2^\#$?

An instrumentation relation whose defining formula is expressed solely in terms of core relations is said to be in *core normal form*. Because there are no circular dependences, an instrumentation relation's defining formula can always be put in core normal form by repeated substitution until only core relations remain. When ψ_p is in core normal form, or has been converted to core normal form, it is possible to determine the value of each instrumentation relation p by evaluating ψ_p in structure $S_{proto}^\#$. We have

$$\begin{aligned}&\text{for each } u_1, \ldots, u_k \in U^{S_1^\#},\\ &\quad \iota^{S_2^\#}(p)(u_1, \ldots, u_k) := [\![\psi_p(v_1, \ldots, v_k)]\!]_3^{S_{proto}^\#}([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]).\end{aligned} \qquad (3)$$

Thus, in principle it is possible to maintain the values of instrumentation relations via Eq. (3). In practice, however, this approach does not work very well.

As observed elsewhere [Sagiv et al. 2002], when working in 3-valued logic, it is usually possible to retain more precision by defining a special *instrumentation-relation-maintenance formula*, $\mu_{p,st}(v_1, \ldots, v_k)$, and evaluating $\mu_{p,st}(v_1, \ldots, v_k)$ in structure $S^{\#}_1$. We have

$$\begin{array}{l}\text{for each } u_1, \ldots, u_k \in U^{S^{\#}_1},\\ \quad \iota^{S^{\#}_2}(p)(u_1, \ldots, u_k) := [\![\mu_{p,st}(v_1, \ldots, v_k)]\!]_3^{S^{\#}_1}([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]).\end{array} \tag{4}$$

The advantage of the relation-maintenance approach is that the results of program analysis can be more accurate: Example 3.2 shows that the relation-maintenance approach enables the precise tracking of “sharing”, information that may be essential for verifying the correctness of list-manipulating procedures. In 3-valued logic, when $\mu_{p,st}$ is defined appropriately, the relation-maintenance strategy can generate a definite value (0 or 1) when the evaluation of ψ_p on $S^{\#}_{proto}$ generates the indefinite value 1/2.

To ensure that an analysis is conservative, however, one must also show that the following property holds.

Definition 3.1. Suppose that p is an instrumentation relation defined by formula ψ_p. Relation-maintenance formula $\mu_{p,st}$ *maintains p correctly for statement st* if, for all $S \in \mathcal{S}_2[\mathcal{R}]$ and all Z, $[\![\mu_{p,st}]\!]_2^S(Z) = [\![\psi_p]\!]_2^{[\![st]\!]_2(S)}(Z)$.

For an instrumentation relation in core normal form, it is always possible to provide a relation-maintenance formula that satisfies Definition 3.1 by defining $\mu_{p,st}$ as

$$\mu_{p,st} \overset{\text{def}}{=} \psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}], \tag{5}$$

where $\varphi[q \hookleftarrow \varphi']$ denotes the formula obtained from φ by replacing each relation occurrence $q(w_1, \ldots, w_k)$ by $\varphi'\{w_1, \ldots, w_k\}$, and $\varphi'\{w_1, \ldots, w_k\}$ denotes the formula obtained from $\varphi'(v_1, \ldots, v_k)$ by replacing each free occurrence of variable v_i by w_i.

The formula $\mu_{p,st}$ defined in Eq. (5) maintains p correctly for statement *st* because, by the 2-valued version of Eq. (2), $[\![\tau_{c,st}]\!]_2^{S_1}(Z) = [\![c]\!]_2^{S_{proto}}(Z)$; consequently, when $\mu_{p,st}$ of Eq. (5) is evaluated in structure S_1, the use of $\tau_{c,st}$ in place of c is equivalent to using the value of c when ψ_p is evaluated in S_{proto}; that is, for all Z, $[\![\psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}]]\!]_2^{S_1}(Z) = [\![\psi_p]\!]_2^{S_{proto}}(Z)$. However—and this is precisely the drawback of using Eq. (5) to obtain the $\mu_{p,st}$—the steps of evaluating $[\![\psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}]]\!]_2^{S_1}(Z)$ *mimic exactly* those of evaluating $[\![\psi_p]\!]_2^{S_{proto}}(Z)$. Consequently, when we pass to 3-valued logic, for all Z, $[\![\psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}]]\!]_3^{S^{\#}_1}(Z)$ yields exactly the same value as $[\![\psi_p]\!]_3^{S^{\#}_{proto}}(Z)$ (i.e., as evaluating Eq. (3)). Thus, although $\mu_{p,st}$ that satisfy Definition 3.1 can be obtained automatically via Eq. (5), this approach does not provide a satisfactory solution to the relation-maintenance problem.

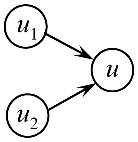


Fig. 11. A store in which u is shared; that is, $is_n(u) = 1$.

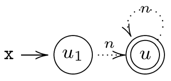
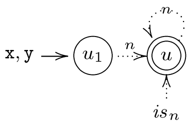
<table>
<tr><td>Structure before</td><td><b>unary rels.</b><table><tr><th>indiv.</th><th>x</th><th>y</th><th>is_n</th></tr><tr><td>u_1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>u</td><td>0</td><td>0</td><td>0</td></tr></table> <b>binary rels.</b><table><tr><th>n</th><th>u_1</th><th>u</th></tr><tr><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><table><tr><th>eq</th><th>u_1</th><th>u</th></tr><tr><td>u_1</td><td>1</td><td>0</td></tr><tr><td>u</td><td>0</td><td>1/2</td></tr></table> </td></tr>
<tr><td>Statement</td><td>y = x</td></tr>
<tr><td>Relation-transfer formulas</td><td>$\tau_{x,\text{y=x}}(v) = x(v)$
$\tau_{y,\text{y=x}}(v) = x(v)$
$\tau_{n,\text{y=x}}(v_1, v_2) = n(v_1, v_2)$
$\tau_{eq,\text{y=x}}(v_1, v_2) = eq(v_1, v_2)$</td></tr>
<tr><td>Relation-maintenance formula</td><td>$\mu_{is_n,\text{y=x}}(v) = \exists\, v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$</td></tr>
<tr><td>Structure after</td><td><b>unary rels.</b><table><tr><th>indiv.</th><th>x</th><th>y</th><th>is_n</th></tr><tr><td>u_1</td><td>1</td><td>1</td><td>0</td></tr><tr><td>u</td><td>0</td><td>0</td><td>1/2</td></tr></table> <b>binary rels.</b><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><table><tr><th>eq</th><th>u_1</th><th>u</th></tr><tr><td>u_1</td><td>1</td><td>0</td></tr><tr><td>u</td><td>0</td><td>1/2</td></tr></table> </td></tr>
</table>

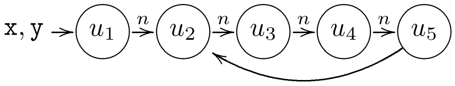
Fig. 12. An illustration of the loss of precision in the value of is_n when its relation-maintenance formula is defined by $\exists\, v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$. The use of this relation-maintenance formula causes a structure to be created in which the individual u may represent a shared memory cell.

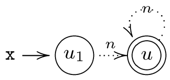
Example 3.2. Eq. (6) shows the defining formula for the instrumentation relation is_n ("is-shared using n fields"),

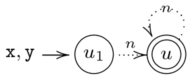
$$is_n(v) \overset{\text{def}}{=} \exists\, v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2, \tag{6}$$

which captures whether a memory cell is pointed to by two or more pointer fields of memory cells, for example, see Figure 11.

Figure 12 illustrates how execution of the statement y = x causes the value of is_n to lose precision when its relation-maintenance formula is created according to Eq. (5). The initial 3-valued structure represents all singly-linked lists of length 2 or more in which all memory cells are unshared. Because execution of y = x does not change the value of core relation n, $\tau_{n,\text{y=x}}(v_1, v_2)$ is $n(v_1, v_2)$, and hence the formula $\mu_{is_n,\text{y=x}}(v)$ created according to Eq. (5) is $\exists\, v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$. As shown in Figure 12, the structure created using this maintenance formula is not as precise as we would like. In particular, $is_n(u) = 1/2$, which means that u can represent a shared cell. Thus, the final 3-valued structure also represents certain cyclic linked lists, such as the following one.



<table>
<tr><td>Structure before</td><td>
<b>unary rels.</b>
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>is_n</th></tr>
<tr><td>u_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></tr>
</table>
<b>binary rels.</b>
<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>
<table>
<tr><th>eq</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>


</td></tr>
<tr><td>Statement</td><td>y = x</td></tr>
<tr><td>Relation-transfer formulas</td><td>$\tau_{x,\mathtt{y=x}}(v) = x(v)$
$\tau_{y,\mathtt{y=x}}(v) = x(v)$
$\tau_{n,\mathtt{y=x}}(v_1, v_2) = n(v_1, v_2)$
$\tau_{eq,\mathtt{y=x}}(v_1, v_2) = eq(v_1, v_2)$</td></tr>
<tr><td>Relation-maintenance formula</td><td>$\mu_{is_n,\mathtt{y=x}}(v) = is_n(v)$</td></tr>
<tr><td>Structure after</td><td>
<b>unary rels.</b>
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>is_n</th></tr>
<tr><td>u_1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>0</td><td>0</td></tr>
</table>
<b>binary rels.</b>
<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>
<table>
<tr><th>eq</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>


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

Fig. 13. Example showing how the imprecision that was illustrated in Figure 12 is avoided with the relation-maintenance formula $\mu_{is_n,\mathtt{y=x}}(v) = is_n(v)$. (Example 5.1 shows how this is generated automatically.)

This sort of imprecision can usually be avoided by devising better relation-maintenance formulas. For instance, when $\mu_{is_n,\mathtt{y=x}}(v)$ is defined to be the formula $is_n(v)$, meaning that y = x does not change the value of $is_n(v)$, the imprecision illustrated in Figure 12 is avoided (see Figure 13). Hand-crafted relation-maintenance formulas for a variety of instrumentation relations are given in Sagiv et al. [2002], Lev-Ami and Sagiv [2000], and TVLA []; however, those formulas were created by ad hoc methods.

To sum up, prior to the work presented in this article, the user needed to supply a formula $\mu_{p,st}$ for each instrumentation relation p and each statement st. In effect, the user needed to write down two *separate* characterizations of each instrumentation relation p: (i) ψ_p, which defines p directly; and (ii) $\mu_{p,st}$, which specifies how execution of each kind of statement in the language affects p. Moreover, it was the user's responsibility to ensure that the two characterizations were mutually consistent. In contrast, with the method for automatically creating relation-maintenance formulas presented in Section 5 and Section 6, the user's responsibility is dramatically reduced: the user only needs to give a *single* characterization of each instrumentation relation p, namely, by defining ψ_p. (In separate work, we have developed ways to use inductive logic programming to *discover* an appropriate set of instrumentation relations that define a suitable abstraction for checking whether a given program has a given property [Loginov et al. 2005, 2007; Loginov 2006].)

4. OUR APPROACH AT AN INFORMAL LEVEL

As illustrated by Example 3.2, relation-maintenance formulas that are defined by Eq. (5) can yield imprecise answers. In essence, Eq. (5) specifies that the new value of instrumentation relation p should be computed using its defining formula ψ_p, but taking into account updates to any core relation c that occurs in ψ_p. Unfortunately, the approach of Eq. (5) is equivalent to that of Eq. (3): they both rely on recomputing the value of instrumentation relation p based on its defining formula ψ_p. In the presence of abstraction, the indefinite value 1/2 for a core relation tuple often causes recomputed

Structure before

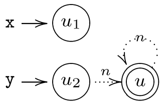
unary rels.

indiv.	x	y	is_n
u_1	1	0	0
u_2	0	1	0
u	0	0	0

binary rels.

n	u_1	u_2	u
u_1	0	0	0
u_2	0	0	1/2
u	0	0	1/2

eq	u_1	u_2	u
u_1	1	0	0
u_2	0	1	0
u	0	0	1/2



Statement

$\mathtt{x \rightarrow n = y}$ (assuming $\mathtt{x \rightarrow n = NULL}$)

Relation-transfer formulas

$$\begin{aligned} \tau_{x,\mathtt{x\rightarrow n=y}}(v) &= x(v) \\ \tau_{y,\mathtt{x\rightarrow n=y}}(v) &= y(v) \\ \tau_{n,\mathtt{x\rightarrow n=y}}(v_1, v_2) &= n(v_1, v_2) \vee (x(v_1) \wedge y(v_2)) \\ \tau_{eq,\mathtt{x\rightarrow n=y}}(v_1, v_2) &= eq(v_1, v_2) \end{aligned}$$

Relation-maintenance formula

$$\mu_{is_n,\mathtt{x\rightarrow n=y}}(v) = is_n(v) \vee (y(v) \wedge \exists\, v_1 : n(v_1, v))$$

Structure after

unary rels.

indiv.	x	y	is_n
u_1	1	0	0
u_2	0	1	0
u	0	0	0

binary rels.

n	u_1	u_2	u
u_1	0	1	0
u_2	0	0	1/2
u	0	0	1/2

eq	u_1	u_2	u
u_1	1	0	0
u_2	0	1	0
u	0	0	1/2

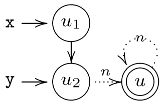


Fig. 14. Example of a nontrivial relation-maintenance formula for relation is_n.

instrumentation-relation tuples to evaluate to 1/2, as well. For instance, as illustrated in Example 3.2, the dashed n edges incident on u in Figure 12 cause $is_n(u)$ to evaluate to 1/2.

As we saw in Figure 13, such recomputation is not always necessary. The values of n (the only core relation that is used to define is_n) cannot change as a result of executing $\mathtt{y = x}$; consequently, the values of instrumentation relation is_n do not need to change as a result of $\mathtt{y = x}$. Moreover, as will become apparent shortly, even when the values of tuples in an instrumentation relation do need to change, they can often be maintained more precisely by means other than recomputation.

The framework of Sagiv et al. includes a mechanism for maintaining more precise values for core relation tuples in abstract structures. Roughly speaking, the *focus* operation is used to ensure that the core relation tuples in the "vicinity" of an update have precise values during a structure transformation, although they may be set to 1/2 when abstraction is applied at the end of the transformation. (For more details about *focus*, see Sagiv et al. [2002, Section 6.3].) Most programming languages have the property that they perform only localized changes to core relations. As a practical matter, what this meant for the TVLA system (prior to our work) was that it was usually possible to create *hand-crafted* relation-maintenance formulas that retain precision under most circumstances. However, prior to the adoption in TVLA of the techniques presented in Section 5 and Section 6 for creating relation-maintenance formulas automatically, the task of crafting a good set of relation-maintenance formulas required substantial expertise, and remained a bit of "black art."

Figure 14 illustrates some of the issues; it addresses the problem of maintaining is_n in response to the execution of the statement $\mathtt{x \rightarrow n = y}$, assuming that $\mathtt{x \rightarrow n = NULL}$. The statement changes relation n: it adds a new n edge from the individual pointed to by $\mathtt{x}$ to that pointed to by $\mathtt{y}$. Thus, the relation-maintenance formula for is_n is nontrivial. However, by noting that is_n can only change in a small part of the structure (the "vicinity" of the update), one can specify the following incremental relation-maintenance formula.

$$\mu_{is_n,\mathtt{x\rightarrow n=y}}(v) = is_n(v) \vee (y(v) \wedge \exists\, v_1 : n(v_1, v)) \qquad (7)$$

φ	$\Delta_{st}[\varphi]$
$\mathbf{1}$	$\mathbf{0}$
$\mathbf{0}$	$\mathbf{0}$
$p(w_1, \ldots, w_k), p \in \mathcal{C}$	$(\tau_{p,st} \oplus p)\{w_1, \ldots, w_k\}$
$p(w_1, \ldots, w_k), p \in \mathcal{I}$	$\Delta_{st}[\psi_p]\{w_1, \ldots, w_k\}$
$\varphi_1 \oplus \varphi_2$	$\Delta_{st}[\varphi_1] \oplus \Delta_{st}[\varphi_2]$
$\varphi_1 \wedge \varphi_2$	$(\Delta_{st}[\varphi_1] \wedge \varphi_2) \oplus (\varphi_1 \wedge \Delta_{st}[\varphi_2]) \oplus (\Delta_{st}[\varphi_1] \wedge \Delta_{st}[\varphi_2])$
$\forall v \colon \varphi_1$	$(\forall v \colon \varphi_1) \mathbin{?} (\exists v \colon \Delta_{st}[\varphi_1]) : (\forall v \colon \varphi_1 \oplus \Delta_{st}[\varphi_1])$

Fig. 15. A finite-differencing scheme for first-order formulas, based on exclusive-or ($\oplus$).

Eq. (7) reuses the *stored* value of is_n for all individuals, except the one that is pointed to by y. For that individual, it checks whether it has an incoming n edge prior to the update. In Figure 14 it does not, and the value of is_n remains 0 for all individuals.

In our first attempt to automate the process of computing incremental relation-maintenance formulas for first-order logic, we defined the finite-differencing scheme shown in Figure 15. In this scheme, $\Delta_{st}[\varphi]$ captures the change to φ's value. With Figure 15, the maintenance formula for instrumentation relation p is

$$\mu_{p,st} \overset{\text{def}}{=} p \oplus \Delta_{st}[\psi_p], \tag{8}$$

where $\oplus$ denotes exclusive-or. However, in 3-valued logic, we have $1/2 \oplus V = 1/2$, regardless of whether V is 0, 1, or 1/2. Consequently, Eq. (8) has the unfortunate property that if $p(u) = 1/2$, then $\mu_{p,st}$ evaluates to 1/2 on u, and $p(u)$ becomes "pinned" to the indefinite value 1/2; it will have the value 1/2 in all successor structures $S^{\#}_2$, in all successors of $S^{\#}_2$, and so on. With Eq. (8), $p(u)$ can never reacquire a definite value.

This led us to consider a scheme that separates the negative change to a formula's value from the positive change. We have

$$\mu_{p,st} = p \mathbin{?} \neg\Delta^-_{st}[\psi_p] : \Delta^+_{st}[\psi_p], \tag{9}$$

where finite-differencing operators $\Delta^-_{st}[\cdot]$ and $\Delta^+_{st}[\cdot]$ capture the negative and positive changes, respectively. (These operators are discussed in detail in Section 5 and Section 6.) In this approach to the relation-maintenance problem, the two finite-differencing operators characterize the tuples of a relation that are subtracted and added in response to a structure transformation.

Because they have the form $p ? \neg\Delta^-_{st}[\psi_p] : \Delta^+_{st}[\psi_p]$, the maintenance formulas created using Eq. (9) do not suffer from the problem exhibited by the maintenance formulas created using Eq. (8) (just discussed). The use of if-then-else allows $p(u)$ to reacquire a definite value after it has been set to 1/2: when $p(u)$ is 1/2, $\mu_{p,st}$ evaluates to a definite value on u if $[\![\Delta^-_{st}[\psi_p(v)]]\!]^{S^{\#}}_3([v \mapsto u])$ is 1 and $[\![\Delta^+_{st}[\psi_p(v)]]\!]^{S^{\#}}_3([v \mapsto u])$ is 0, or vice versa.

Limitations. The finite-differencing technique that we present in this article is applicable to any method in which systems are described as evolving (2-valued or 3-valued) logical structures. However, it is important to note some limitations of the approach. First, it relies on a first-order encoding of all properties. In particular, the finite-differencing technique includes no explicit handling of numerical properties; we expect those to be modeled implicitly by other relations, such as the binary relation *dle* (see Section 2.2). It may be possible to combine numerical finite differencing [Goldstine 1977] with our approach, thus creating a finite-differencing technique that is prepared to handle numerical properties explicitly. Second, for maintaining transitive closure (reachability) relations, the finite-differencing technique is effective only when transitive closure relations can be updated using first-order formulas. Section 6 describes in detail the extent to which our approach can be used to maintain transitive closure relations.

5. RELATION MAINTENANCE FOR 2-VALUED (AND 3-VALUED) FIRST-ORDER LOGIC VIA FINITE DIFFERENCING

This section presents an algorithm for creating relation-maintenance formulas that is based on finite differencing. The discussion will be couched primarily in terms of 2-valued logic; however, by the embedding theorem (Theorem 2.6, [Sagiv et al. 2002, Theorem 4.9]), the relation-maintenance formulas that we derive provide sound results when interpreted in 3-valued logic. In 3-valued logic, as demonstrated in Figure 13 (and discussed further in Example 5.1), the resulting formula can lead to a strictly more precise result than merely reevaluating an instrumentation relation's defining formula.

Our algorithm for creating a relation-maintenance formula $\mu_{p,st}$, for $p \in \mathcal{I}$, uses an incremental computation strategy: $\mu_{p,st}$ is defined in terms of the stored (prestate) value of p, along with two finite-differencing operators, denoted by $\Delta^-_{st}[\cdot]$ and $\Delta^+_{st}[\cdot]$. The finite-differencing operators capture the negative and positive changes, respectively, that execution of structure transformer *st* induces in an instrumentation relation's value. The formula $\mu_{p,st}$ is defined as follows.

$$\mu_{p,st} = p \; ? \; \neg\Delta^-_{st}[\psi_p] : \Delta^+_{st}[\psi_p] \qquad (10)$$

Maintenance formula $\mu_{p,st}$ specifies the new value of p (i.e., its value in S_2, in the case of a 2-valued structure, or $S^\#_2$, in the case of a 3-valued structure) in terms of the old values of p, $\Delta^-_{st}[\psi_p]$, and $\Delta^+_{st}[\psi_p]$ (i.e., their values in S_1 or $S^\#_1$). Eq. (10) states that if p's old value is 1, then its new value is 1 unless there is a negative change; if p's old value is 0, then its new value is 1 if there is a positive change.

Figure 16 depicts how the static analysis engine evaluates $\Delta^-_{st}[\psi_p]$ and $\Delta^+_{st}[\psi_p]$ in $S^\#_1$ and combines these values with the old value p to obtain the desired new value p''. The operators $\Delta^-_{st}[\cdot]$ and $\Delta^+_{st}[\cdot]$ are defined recursively, as shown in Figure 17. The definitions in Figure 17 make use of the operator $\mathbf{F}_{st}[\varphi]$ (standing for "Future"), defined as follows.

$$\mathbf{F}_{st}[\varphi] \overset{\text{def}}{=} \varphi \; ? \; \neg\Delta^-_{st}[\varphi] : \Delta^+_{st}[\varphi] \qquad (11)$$

Thus, maintenance formula $\mu_{p,st}$ can also be expressed as $\mu_{p,st} = \mathbf{F}_{st}[p]$.

Formula (11) and Figure 17 define a syntax-directed translation scheme that can be implemented via a recursive walk over a formula φ. The operators

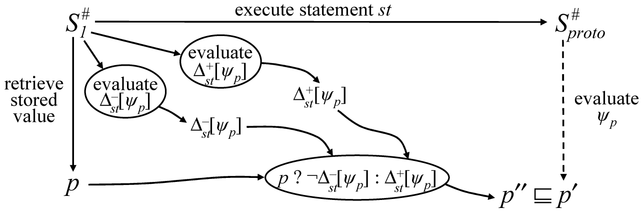


Fig. 16. How to maintain the value of ψ_p in 3-valued logic in response to changes in the values of core relations caused by the execution of structure transformer st.

$\Delta_{st}^-[\cdot]$ and $\Delta_{st}^+[\cdot]$ are mutually recursive. For instance, $\Delta_{st}^+[\neg\varphi_1] = \Delta_{st}^-[\varphi_1]$ and $\Delta_{st}^-[\neg\varphi_1] = \Delta_{st}^+[\varphi_1]$. Moreover, each occurrence of $\mathbf{F}_{st}[\varphi_i]$ contains additional occurrences of $\Delta_{st}^-[\varphi_i]$ and $\Delta_{st}^+[\varphi_i]$.

Note how $\Delta_{st}^-[\cdot]$ and $\Delta_{st}^+[\cdot]$ for $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$ exhibit the "convolution" pattern characteristic of differentiation, finite differencing, and divided differencing.

Continuing the analogy with differentiation, it helps to bear in mind that the "independent variables" are the core relations—which are being changed by the $\tau_{c,st}$ formulas; the dependent variable is the value of φ. A formal justification of Figure 17 is stated later (Theorem 5.3 and Corrollary 5.4); here we merely explain informally a few of the cases from Figure 17.

$\Delta_{st}^+[\mathbf{1}] = \mathbf{0}$, $\Delta_{st}^-[\mathbf{1}] = \mathbf{0}$. The value of atomic formula $\mathbf{1}$ does not depend on any core relations; hence its value is unaffected by changes in them.

$\Delta_{st}^-[\varphi_1 \wedge \varphi_2] = (\Delta_{st}^-[\varphi_1] \wedge \varphi_2) \vee (\varphi_1 \wedge \Delta_{st}^-[\varphi_2])$. Tuples of individuals removed from $\varphi_1 \wedge \varphi_2$ are either tuples of individuals removed from φ_1 for which φ_2 also holds (i.e., $(\Delta_{st}^-[\varphi_1] \wedge \varphi_2)$), or they are tuples of individuals removed from φ_2 for which φ_1 also holds, (i.e., $(\varphi_1 \wedge \Delta_{st}^-[\varphi_2])$.

$\Delta_{st}^+[\exists\, v\colon \varphi_1] = (\exists\, v\colon \Delta_{st}^+[\varphi_1]) \wedge \neg(\exists\, v\colon \varphi_1)$. For $\exists\, v\colon \varphi_1$ to change value from 0 to 1, there must be at least one individual for which φ_1 changes value from 0 to 1 (i.e., $\exists\, v\colon \Delta_{st}^+[\varphi_1]$ holds), and $\exists\, v\colon \varphi_1$ must not already hold (i.e., $\neg(\exists\, v\colon \varphi_1)$ holds).

$\Delta_{st}^+[p(w_1, \ldots, w_k)] = (\exists\, v\colon \Delta_{st}^+[\varphi_1]) \wedge \neg p$, if $p \in \mathcal{I}$ and $\psi_p \equiv \exists\, v\colon \varphi_1$. This case is similar to the previous one, except that the term to ensure that $\exists\, v\colon \varphi_1$ does not already hold (i.e., $\neg(\exists\, v\colon \varphi_1)$) is replaced by the formula $\neg p$. Thus, when $(\exists\, v\colon \Delta_{st}^+[\varphi_1]) \wedge \neg p$ is evaluated, the stored value of $\exists\, v\colon \varphi_1$, that is, p, will be used instead of the value obtained by reevaluating $\exists\, v\colon \varphi_1$.

$\Delta_{st}^+[p(w_1, \ldots, w_k)] = \Delta_{st}^+[\psi_p\{w_1, \ldots, w_k\}]$, if $p \in \mathcal{I}$ and $\psi_p \not\equiv \exists\, v\colon \varphi_1$. To characterize the positive changes to p, apply Δ_{st}^+ to p's defining formula ψ_p.

One special case is also worth noting: $\Delta_{st}^+[v_1 = v_2] = \mathbf{0}$ and $\Delta_{st}^-[v_1 = v_2] = \mathbf{0}$ because the value of the atomic formula $(v_1 = v_2)$ (shorthand for $eq(v_1, v_2)$) does not depend on any core relations; hence, its value is unaffected by changes in them.[5]

[5] We avoid issues that could arise due to changes in a structure's universe of individuals by modeling storage allocation and deallocation via a free-storage list. We describe our solution in more detail at the end of this section.

φ	$\Delta^{+}_{st}[\varphi]$	$\Delta^{-}_{st}[\varphi]$
$\mathbf{1}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$p(w_1,\ldots,w_k)$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p \mathbin{?} \neg\delta^{-}_{p,st} : \delta^{+}_{p,st}$	$(\delta^{+}_{p,st} \wedge \neg p)\{w_1,\ldots,w_k\}$	$(\delta^{-}_{p,st} \wedge p)\{w_1,\ldots,w_k\}$
$p(w_1,\ldots,w_k)$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p \vee \delta_{p,st}$ or $\delta_{p,st} \vee p$	$(\delta_{p,st} \wedge \neg p)\{w_1,\ldots,w_k\}$	$\mathbf{0}$
$p(w_1,\ldots,w_k)$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p \wedge \delta_{p,st}$ or $\delta_{p,st} \wedge p$	$\mathbf{0}$	$(\neg\delta_{p,st} \wedge p)\{w_1,\ldots,w_k\}$
$p(w_1,\ldots,w_k)$, $p \in \mathcal{C}$, but $\tau_{p,st}$ is not of the above forms	$(\tau_{p,st} \wedge \neg p)\{w_1,\ldots,w_k\}$	$(p \wedge \neg\tau_{p,st})\{w_1,\ldots,w_k\}$
$p(w_1,\ldots,w_k)$, $p \in \mathcal{I}$	$((\exists\, v\colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg p)\{w_1,\ldots,w_k\}$ if $\psi_p \equiv \exists\, v\colon \varphi_1$ $\Delta^{+}_{st}[\psi_p]\{w_1,\ldots,w_k\}$ otherwise	$((\exists\, v\colon \Delta^{-}_{st}[\varphi_1]) \wedge p)\{w_1,\ldots,w_k\}$ if $\psi_p \equiv \forall\, v\colon \varphi_1$ $\Delta^{-}_{st}[\psi_p]\{w_1,\ldots,w_k\}$ otherwise
$\neg\varphi_1$	$\Delta^{-}_{st}[\varphi_1]$	$\Delta^{+}_{st}[\varphi_1]$
$\varphi_1 \vee \varphi_2$	$(\Delta^{+}_{st}[\varphi_1] \wedge \neg\varphi_2) \vee (\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_2])$	$(\Delta^{-}_{st}[\varphi_1] \wedge \neg\mathbf{F}_{st}[\varphi_2]) \vee (\neg\mathbf{F}_{st}[\varphi_1] \wedge \Delta^{-}_{st}[\varphi_2])$
$\varphi_1 \wedge \varphi_2$	$(\Delta^{+}_{st}[\varphi_1] \wedge \mathbf{F}_{st}[\varphi_2]) \vee (\mathbf{F}_{st}[\varphi_1] \wedge \Delta^{+}_{st}[\varphi_2])$	$(\Delta^{-}_{st}[\varphi_1] \wedge \varphi_2) \vee (\varphi_1 \wedge \Delta^{-}_{st}[\varphi_2])$
$\exists\, v\colon \varphi_1$	$(\exists\, v\colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg(\exists\, v\colon \varphi_1)$	$(\exists\, v\colon \Delta^{-}_{st}[\varphi_1]) \wedge \neg(\exists\, v\colon \mathbf{F}_{st}[\varphi_1])$
$\forall\, v\colon \varphi_1$	$(\exists\, v\colon \Delta^{+}_{st}[\varphi_1]) \wedge (\forall\, v\colon \mathbf{F}_{st}[\varphi_1])$	$(\exists\, v\colon \Delta^{-}_{st}[\varphi_1]) \wedge (\forall\, v\colon \varphi_1)$

Fig. 17. Finite-difference formulas for first-order formulas.

$$\Delta^{+}_{st}[is_n(v)] \;=\; \left(\exists\, v_1, v_2 : \left(\begin{array}{rl} & (\Delta^{+}_{st}[n(v_1,v)] \land \mathbf{F}_{st}[n(v_2,v)]) \\ \lor & (\mathbf{F}_{st}[n(v_1,v)] \land \Delta^{+}_{st}[n(v_2,v)]) \end{array}\right) \land v_1 \neq v_2\right) \land \neg is_n(v)$$

$$\Delta^{-}_{st}[is_n(v)] \;=\; \left\{\begin{array}{l} \quad \left(\exists\, v_1, v_2 : \left(\begin{array}{rl} & (\Delta^{-}_{st}[n(v_1,v)] \land n(v_2,v)) \\ \lor & (n(v_1,v) \land \Delta^{-}_{st}[n(v_2,v)]) \end{array}\right) \land v_1 \neq v_2\right) \\ \land \\ \qquad \neg \left(\exists\, v_1, v_2 : \left(\begin{array}{l} (n(v_1,v) \land n(v_2,v) \land v_1 \neq v_2) \\ \quad ?\ \neg \left(\left(\begin{array}{rl} & (\Delta^{-}_{st}[n(v_1,v)] \land n(v_2,v)) \\ \lor & (n(v_1,v) \land \Delta^{-}_{st}[n(v_2,v)]) \end{array}\right) \land v_1 \neq v_2\right) \\ \quad :\ \left(\begin{array}{rl} & (\Delta^{+}_{st}[n(v_1,v)] \land \mathbf{F}_{st}[n(v_2,v)]) \\ \lor & (\mathbf{F}_{st}[n(v_1,v)] \land \Delta^{+}_{st}[n(v_2,v)]) \end{array}\right) \land v_1 \neq v_2 \end{array}\right)\right) \end{array}\right.$$

Fig. 18. Finite-difference formulas for the instrumentation relation $is_n(v)$.

Example 5.1. Consider the instrumentation relation is_n ("is-shared using n fields"), defined in Eq. (6) . Figure 18 shows the formulas obtained for $\Delta^{+}_{st}[is_n(v)]$ and $\Delta^{-}_{st}[is_n(v)]$.

For a particular statement, the formulas in Figure 18 can usually be simplified. For instance, for $\mathrm{y} = \mathrm{x}$, the relation-transfer formula $\tau_{n,\mathrm{y=x}}(v_1, v_2)$ is $n(v_1, v_2)$; see Figure 12. Thus, by Figure 17, the formulas for $\Delta^{-}_{\mathrm{y=x}}[n(v_1, v)]$ and $\Delta^{+}_{\mathrm{y=x}}[n(v_1, v)]$ are both $n(v_1, v) \land \neg n(v_1, v)$, which simplifies to $\mathbf{0}$. (In our implementation, simplifications are performed greedily at formula construction time; e.g., the constructor for $\land$ rewrites $\mathbf{0} \land p$ to $\mathbf{0}$, $\mathbf{1} \land p$ to p, $p \land \neg p$ to $\mathbf{0}$, etc.) The formulas in Figure 18 simplify to $\Delta^{+}_{\mathrm{y=x}}[is_n(v)] = \mathbf{0}$ and $\Delta^{-}_{\mathrm{y=x}}[is_n(v)] = \mathbf{0}$. Consequently, $\mu_{is_n,\mathrm{y=x}}(v) = \mathbf{F}_{\mathrm{y=x}}[is_n(v)] = is_n(v)\ ?\ \neg\mathbf{0} : \mathbf{0} = is_n(v)$. As shown in Figure 13, this definition of $\mu_{is_n,\mathrm{y=x}}(v)$ avoids the imprecision that was illustrated in Example 3.2.

Correctness of the relation-maintenance scheme. The correctness of the finite-differencing scheme given before is established with the help of the following lemma.

LEMMA 5.2. *For every formula φ, φ_1, φ_2 and structure transformer st, the following properties hold:*[6]

(i) $\Delta^{+}_{st}[\varphi] \overset{\text{meta}}{\Longleftrightarrow} \boldsymbol{F}_{st}[\varphi] \land \neg\varphi$

(ii) $\Delta^{-}_{st}[\varphi] \overset{\text{meta}}{\Longleftrightarrow} \varphi \land \neg\boldsymbol{F}_{st}[\varphi]$

(iii) (a) $\boldsymbol{F}_{st}[\neg\varphi_1] \overset{\text{meta}}{\Longleftrightarrow} \neg\boldsymbol{F}_{st}[\varphi_1]$

(b) $\boldsymbol{F}_{st}[\varphi_1 \lor \varphi_2] \overset{\text{meta}}{\Longleftrightarrow} \boldsymbol{F}_{st}[\varphi_1] \lor \boldsymbol{F}_{st}[\varphi_2]$

(c) $\boldsymbol{F}_{st}[\varphi_1 \land \varphi_2] \overset{\text{meta}}{\Longleftrightarrow} \boldsymbol{F}_{st}[\varphi_1] \land \boldsymbol{F}_{st}[\varphi_2]$

(d) $\boldsymbol{F}_{st}[\exists\, v : \varphi_1] \overset{\text{meta}}{\Longleftrightarrow} \exists\, v : \boldsymbol{F}_{st}[\varphi_1]$

(e) $\boldsymbol{F}_{st}[\forall\, v : \varphi_1] \overset{\text{meta}}{\Longleftrightarrow} \forall\, v : \boldsymbol{F}_{st}[\varphi_1]$

PROOF. See electronic Appendix A available in the ACM Digital Library. □

[6] To simplify the presentation, we use $lhs \overset{\text{meta}}{\Longleftrightarrow} rhs$ and $lhs \overset{\text{meta}}{\Longrightarrow} rhs$ as shorthands for $[\![lhs]\!]^S_2(Z) = [\![rhs]\!]^S_2(Z)$ and $[\![lhs]\!]^S_2(Z) \leq [\![rhs]\!]^S_2(Z)$, respectively, for any $S \in \mathcal{S}_2$ and assignment Z that is complete for lhs and rhs.

Lemma 5.2 shows that for structures in $\mathcal{S}_2$, $\Delta^+_{st}[\varphi]$ specifies the tuples that are not in the relation defined by φ, but need to be added in response to the execution of *st*, and that $\Delta^-_{st}[\varphi]$ specifies the tuples that are in the relation defined by φ that need to be removed. Lemma 5.2 is used in the proof of the following theorem, which ensures the correctness of the finite-differencing transformation given in Figure 17, as well as the finite-differencing-based scheme for relation maintenance given in Eq. (10).

THEOREM 5.3. *Let S_1 be a structure in $\mathcal{S}_2$, and let S_{proto} be the proto-structure obtained from S_1 using structure transformer st. Let S_2 be the structure obtained by using S_{proto} as the first approximation to S_2 and then filling in instrumentation relations in a topological ordering of the dependences among them: for each arity-k relation $p \in \mathcal{I}$, $\iota^{S_2}(p)$ is obtained by evaluating $[\![\psi_p(v_1, \ldots, v_k)]\!]_2^{S_2}([v_1 \mapsto u'_1, \ldots, v_k \mapsto u'_k])$ for all tuples $(u'_1, \ldots, u'_k) \in (U^{S_2})^k$. Then for every formula $\varphi(v_1, \ldots, v_k)$ and complete assignment Z for $\varphi(v_1, \ldots, v_k)$, $[\![\boldsymbol{F}_{st}[\varphi(v_1, \ldots, v_k)]]\!]_2^{S_1}(Z) = [\![\varphi(v_1, \ldots, v_k)]\!]_2^{S_2}(Z)$.*

PROOF. See electronic Appendix A in the ACM Digital Library. □

For structures in $\mathcal{S}_3$, the soundness of the finite-differencing transformation given in Figure 17, as well as the finite-differencing-based scheme for relation maintenance given in Eq. (10), follows from Theorem 5.3 by the embedding theorem (Theorem 2.6).

COROLLARY 5.4. *Let $S_1, S_2 \in \mathcal{S}_2$ be defined as in Theorem 5.3. Let $S_1^\# \in \mathcal{S}_3$ be such that $f \colon U^{S_1} \to U^{S_1^\#}$ embeds S_1 in $S_1^\#$, that is, $S_1 \sqsubseteq^f S_1^\#$. Then for every formula $\varphi(v_1, \ldots, v_k)$ and complete assignment Z for $\varphi(v_1, \ldots, v_k)$, $[\![\boldsymbol{F}_{st}[\varphi(v_1, \ldots, v_k)]]\!]_3^{S_1^\#}(f \circ Z) \sqsupseteq [\![\varphi(v_1, \ldots, v_k)]\!]_2^{S_2}(Z)$.*

Optimized formulas for $\boldsymbol{F}_{st}[\varphi]$. For a nonatomic formula φ, the operator $\mathbf{F}_{st}[\varphi]$ defined in formula (11) introduces a copy of φ, because it has no way, in general, to refer to a relation that holds the stored value of φ. The reevaluation of φ inherent in the version of $\mathbf{F}_{st}[\cdot]$ from formula (11) (i.e., $\mathbf{F}_{st}[\varphi] \stackrel{\text{def}}{=} \varphi \,?\, \ldots : \ldots$) may cause a substantial loss of precision. One way to retain higher precision is to propagate $\mathbf{F}_{st}[\cdot]$ into the subformulas of φ, down to the level of atomic formulas (either core relation symbols or instrumentation-relation symbols) as shown in Figure 19.

Suppose that φ' is the result of $\mathbf{F}_{st}[\varphi]$ by the method of Figure 19. An evaluation of φ' will evaluate (copies of) the operators of φ, down to the level of each atomic subformula $p(w_1, \ldots, w_k)$ in φ. At that level, if $p \in \mathcal{I}$, $\mathbf{F}_{st}[\cdot]$ will have introduced an occurrence of p in φ'.

$$\mathbf{F}_{st}[p(w_1, \ldots, w_k)] \stackrel{\text{def}}{=} p(w_1, \ldots, w_k) \,?\, \neg\Delta^-_{st}[p(w_1, \ldots, w_k)] : \Delta^+_{st}[p(w_1, \ldots, w_k)] \tag{12}$$

The occurrence of p in the test refers to the stored ("prestate") value of instrumentation relation p; consequently, the stored tuples of relation p will be used when evaluating φ'.

Note that $\Delta^+_{st}[p]$ and $\Delta^-_{st}[p]$ in formula (12) dispatch according to the case for $p \in \mathcal{I}$ in Figure 17. In particular, because $\mathbf{F}_{st}[\cdot]$ occurs in four of the eight cases

φ	$\mathbf{F}_{st}[\varphi]$
$\mathbf{1}$	$\mathbf{1}$
$\mathbf{0}$	$\mathbf{0}$
$p(w_1,\ldots,w_k), p \in \mathcal{C}$	$\tau_{p,st}\{w_1,\ldots,w_k\}$
$p(w_1,\ldots,w_k), p \in \mathcal{I}$	$p(w_1,\ldots,w_k)\ ?\ \neg\Delta^-_{st}[p(w_1,\ldots,w_k)] : \Delta^+_{st}[p(w_1,\ldots,w_k)]$
$\neg\varphi_1$	$\neg\mathbf{F}_{st}[\varphi_1]$
$\varphi_1 \vee \varphi_2$	$\mathbf{F}_{st}[\varphi_1] \vee \mathbf{F}_{st}[\varphi_2]$
$\varphi_1 \wedge \varphi_2$	$\mathbf{F}_{st}[\varphi_1] \wedge \mathbf{F}_{st}[\varphi_2]$
$\exists\, v\colon \varphi_1$	$\exists\, v\colon \mathbf{F}_{st}[\varphi_1]$
$\forall\, v\colon \varphi_1$	$\forall\, v\colon \mathbf{F}_{st}[\varphi_1]$

Fig. 19. Optimized formulas for the operator $\mathbf{F}_{st}[\varphi]$.

for $\Delta^+_{st}[\cdot]$ and $\Delta^-_{st}[\cdot]$ in Figure 17 (i.e., for $\vee$, $\wedge$, $\exists$, and $\forall$) the optimized $\mathbf{F}_{st}[\cdot]$ is invoked recursively on various subterms of ψ_p.

The correctness of the version of $\mathbf{F}_{st}[\cdot]$ defined in Figure 19 follows from Lemma 5.2.

The method described earlier also usually produces smaller instrumentation-relation-maintenance formulas, and hence creates abstract transformers that generally can be evaluated more quickly. This technique is incorporated into our implementation.

Complexity of the relation-maintenance scheme. There are at most three operations that can be applied to each subformula φ considered during the method for creating a relation-maintenance formula: $\mathbf{F}_{st}[\varphi]$, $\Delta^-_{st}[\varphi]$, and $\Delta^+_{st}[\varphi]$. Duplicate work can be avoided be performing function caching (also known as memoization [Michie 1968]). Moreover, for each of the possible operator node kinds for φ's outermost operator, each of the operations $\mathbf{F}_{st}[\varphi]$, $\Delta^-_{st}[\varphi]$, and $\Delta^+_{st}[\varphi]$ introduces a constant number of operator nodes into the answer formula (together with the results of additional calls on $\mathbf{F}_{st}[\varphi_i]$, $\Delta^-_{st}[\varphi_i]$, and $\Delta^+_{st}[\varphi_i]$ for various subformula(s) φ_i of φ). We will assume that the algorithm (i) uses a DAG representation of the output formula (so that subformulas are shared in the answer formula), (ii) memoizes calls on $\mathbf{F}_{st}[\varphi]$, $\Delta^-_{st}[\varphi]$, and $\Delta^+_{st}[\varphi]$ (thereby possibly creating shared subformulas in the answer formula), and (iii) the hashed-lookup operation used during memoization takes unit time per lookup. Under these assumptions, for an instrumentation relation defined by formula ψ, the overall cost of creating a relation-maintenance formula via our finite-differencing scheme is linear in the size of the DAG that represents ψ in core normal form (see Section 3).

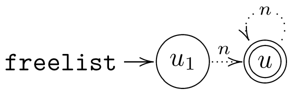
Discussion. Earlier in the article we touted the advantages of being able to apply related 2-valued and 3-valued interpretation functions to a single formula, which, in essence, uses overloading to define two related meaning functions. Thus, it may seem somewhat inconsistent for us to address the problem of maintaining instrumentation relations by an approach that involves explicit transformations of formulas rather than by an approach based on overloading. (In unpublished work, we have studied such an approach—e.g., interpreting instrumentation predicate p's defining formula ψ_p with respect to both the prestate structure S_1 and a specification of the differences between the core predicates of S_1 and those of S_{proto}.) The reason that we use a transformation-based approach is that it gives us an opportunity to simplify the resulting formulas (either on-the-fly, or in a postprocessing phase after finite differencing).

In the context of evaluation in 3-valued logic, simplification is important because even formulas that are tautologies in 2-valued logic may evaluate to 1/2 in 3-valued logic. For instance, $p \vee \neg p$ yields 1/2 when p has the value 1/2, even when p is a nullary relation symbol. The finite-differencing transformation that we implemented uses a formula minimization procedure for 3-valued logic that we developed [Reps et al. 2002]. The minimization procedure applies to propositional logic; for propositional logic, it is guaranteed to return an answer that captures the formula's "supervaluational meaning" [van Fraassen 1966]. This procedure is used as a subroutine in a heuristic method for minimizing first-order formulas; the method works on a formula bottom-up, applying the propositional minimizer to the body of each nonpropositional operator (i.e., each quantifier or transitive closure operator).

A relation-maintenance formula that has been simplified in this way can sometimes yield a definite value in situations where the evaluation of the unsimplified relation-maintenance formula, or, equivalently, an overloaded evaluation of the relation's defining formula, yields 1/2. (For instance, minimizing $p \vee \neg p$ yields 1, which evaluates to 1 even when p has the value 1/2.) Consequently, the formula transformation approach to the relation-maintenance problem leads to more precise static analysis algorithms.

Malloc and free. In Sagiv et al. [2002], the modeling of storage allocation/deallocation operations is carried out with a two-stage structure transformer, the first stage of which changes the number of individuals in the structure. This creates some problems for the finite-differencing approach in establishing appropriate, mutually consistent values for relation tuples that involve the newly allocated individual. Such relation values are needed for the second stage, in which relation-transfer formulas for core relations and relation-maintenance formulas for instrumentation relations are applied in the usual fashion, using Eqs. (2) and (4).

However, there is a simple way to sidestep this problem, which is to model the free-storage list explicitly, making the following substructure part of every 3-valued structure.



(13)

A `malloc` is modeled by advancing the pointer `freelist` into the list, and returning the memory cell that it formerly pointed to. A `free` is modeled by inserting, at the head of `freelist`'s list, the cell being deallocated. This approach models

p	**IntendedMeaning**	ψ_p
$t_n(v_1, v_2)$	Is v_2 reachable from v_1 along `n` fields?	$n^*(v_1, v_2)$
$r_{n,z}(v)$	Is v reachable from pointer variable `z` along `n` fields?	$\exists\, v_1 : z(v_1) \wedge t_n(v_1, v)$
$c_n(v)$	Is v on a directed cycle of `n` fields?	$\exists\, v_1 : n(v_1, v) \wedge t_n(v, v_1)$

Fig. 20. Defining formulas of some instrumentation relations that depend on **RTC**. (Recall that $n^*(v_1, v_2)$ is a shorthand for $(\textbf{RTC}\ v'_1, v'_2 : n(v'_1, v'_2))(v_1, v_2)$.)

limits on available storage naturally, while the introduction of one integrity constraint enables it to model unbounded storage.[7]

It is true that the use of structure (13) to model storage allocation/deallocation operations also causes the number of individuals in a 3-valued structure to change; however, because the new individual is materialized using the usual mechanisms from Sagiv et al. [2002] (namely, the *focus* and *coerce* operations), values for relation tuples that involve the newly materialized individual will always have safe, mutually consistent values.

6. MAINTENANCE FORMULAS FOR REACHABILITY AND TRANSITIVE CLOSURE

Several instrumentation relations that depend on **RTC** are shown in Figure 20. Unfortunately, finding a good way to maintain instrumentation relations defined using **RTC** is challenging because the evaluation of a formula that uses the **RTC** operator in a 3-valued structure generally produces many tuples with the value 1/2. This happens because in an *abstracted* binary relation, tuples ("edges") that involve summary individuals often have the value 1/2. (For instance, see the dashed n edges incident on u in structure (13).) Because the semantics of a tuple (u_1, u_2) computed via **RTC** is defined to be the "max over all paths P from u_1 to u_2 of the minimum value of an edge along P" (see Definitions 2.2 and 2.4), the presence of indefinite edge tuples often causes the path tuple computed for a pair (u_1, u_2) to have the value 1/2. Moreover, it is not known, in general, whether it is possible to write a first-order formula (i.e., without using a transitive closure operator) that specifies how to maintain the closure of a directed graph in response to edge insertions and deletions. Thus, our strategy has been to investigate special cases for classes of instrumentation relations for which first-order maintenance formulas do exist. Whenever these do not apply, the system falls back on safe maintenance formulas (which themselves use **RTC**).

In this section, we confine ourselves to important special cases for the maintenance of instrumentation relations specified via the **RTC** of a binary formula $\varphi_1(v_1, v_2)$. In Section 6.1, we consider the case in which $\varphi_1(v_1, v_2)$ defines a directed acyclic graph. In Section 6.2 we consider the case in which $\varphi_1(v_1, v_2)$ defines a tree-shaped graph. Finally, in Section 6.3, we consider the case in

[7] Instead of a free-storage list, one could use a (bounded or unbounded) *set* of memory locations to model storage allocation and deallocation. In that approach, instead of reachability from the pointer `freelist`, a core unary relation would mark free cells, thus distinguishing them from allocated cells that have been leaked.

which $\varphi_1(v_1, v_2)$ defines a *deterministic graph*; that is, a possibly-cyclic graph, in which every node has outdegree at most one (this class of graphs corresponds to possibly-cyclic linked lists). This collection of techniques allows us to handle most common data structures, such as lists (singly- and doubly-linked; cyclic and acyclic) and trees. The precision of all of these techniques is due to the fact that maintenance of **RTC** after unit-size changes (single-edge additions or deletions)[8] is performed via first-order logical formulas only. However, maintaining **RTC** of an arbitrary directed graph, as well as maintaining **RTC** of restricted classes of graphs with arbitrary-size changes, is not known to be first-order expressible. In such cases, our algorithm returns a formula that uses the **RTC** operator; the evaluation of such a formula may yield more indefinite answers than necessary.

To specify that the maintenance of binary relation $p(v_1, v_2)$ defined as the **RTC** of binary formula $\varphi_1(v_1, v_2)$ should rely on one of the special cases, the user annotates formula φ_1 with attributes. To state that $\varphi_1(v_1, v_2)$ defines a directed acyclic graph, the user gives φ_1 attribute "acyclic"; to state that $\varphi_1(v_1, v_2)$ defines a tree-shaped graph, the user gives φ_1 attribute "tree"; to state that $\varphi_1(v_1, v_2)$ defines a deterministic graph, the user gives φ_1 attribute "function". See Figure 10 for the intended meanings of these attributes.

The analysis uses the attributes to generate integrity constraints to be enforced by the sharpening operation *coerce* (see Section 2.2.2). For instance, when relation $p(v_1, v_2)$ is defined as the **RTC** of formula $\varphi_1(v_1, v_2)$ that is annotated with the attribute "acyclic," the analysis generates the following two constraints.

$$\forall\, v_1, v_2 \colon p(v_1, v_2) \wedge p(v_2, v_1) \;\Rightarrow\; v_1 = v_2$$
$$\forall\, v_1, v_2 \colon p(v_1, v_2) \wedge v_1 \neq v_2 \;\Rightarrow\; \neg p(v_2, v_1)$$

When $\varphi_1(v_1, v_2)$ is annotated with the attribute "tree," the analysis generates the above acyclicity constraints, together with constraints that ensure that $\varphi_1(v_1, v_2)$ is an inverse partial function.

Whenever *coerce* determines that a constraint is (possibly) not satisfied after the application of a transformer, a warning is generated.

6.1 Transitive Closure Maintenance in Directed Acyclic Graphs

Consider a binary instrumentation relation p, defined by $\psi_p(v_1, v_2) \equiv (\mathbf{RTC}\ v_1', v_2' \colon \varphi_1)(v_1, v_2)$. If the graph defined by φ_1 is acyclic, it is possible to give a first-order formula that maintains p after the addition or deletion of a single φ_1-edge. The method we use is a minor modification of a method for maintaining nonreflexive transitive closure in a directed acyclic graph, due to Dong and Su [2000].

In the case of an insertion of a single φ_1-edge, the maintenance formula is

$$\mathbf{F}_{st}[p](v_1, v_2) = p(v_1, v_2) \vee (\exists\, v_1', v_2' \colon p(v_1, v_1') \wedge \Delta^{+}_{st}[\varphi_1](v_1', v_2') \wedge p(v_2', v_2)). \quad (14)$$

[8] These techniques can be extended to handle bounded-size addition and deletion sets.

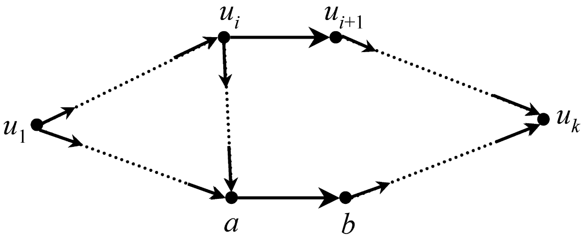


Fig. 21. Edge (a, b) is being deleted; u_i is the last node along path $u_1, \ldots, u_i, u_{i+1}, \ldots, u_k$ from which a is reachable.

The new value of p contains the old tuples of p, as well as those that represent two old paths (i.e., $p(v_1, v'_1)$ and $p(v'_2, v_2)$) connected with the new φ_1-edge (i.e., $\Delta^+_{st}[\varphi_1](v'_1, v'_2)$).

The maintenance formula to handle the deletion of a single φ_1-edge is a bit more complicated. We first identify the tuples of p that represent paths that might rely on the edge to be deleted, and thus may need to be removed from p (S stands for *suspicious*).

$$S[p, \varphi_1](v_1, v_2) = \exists\, v'_1, v'_2 \colon p(v_1, v'_1) \wedge \Delta^-_{st}[\varphi_1](v'_1, v'_2) \wedge p(v'_2, v_2)$$

We next collect a set of p-tuples that definitely remain in p (T stands for *trusted*).

$$T[p, \varphi_1](v_1, v_2) = (p(v_1, v_2) \wedge \neg S[p, \varphi_1](v_1, v_2)) \vee \mathbf{F}_{st}[\varphi_1](v_1, v_2) \qquad (15)$$

Finally, the maintenance formula for p for a single φ_1-edge deletion is

$$\mathbf{F}_{st}[p](v_1, v_2) = \exists\, v'_1, v'_2 \colon T[p, \varphi_1](v_1, v'_1) \wedge T[p, \varphi_1](v'_1, v'_2) \wedge T[p, \varphi_1](v'_2, v_2). \quad (16)$$

Maintenance formulas (14) and (16) maintain p when two conditions hold: the graph defined by φ_1 is acyclic, and the change to the graph is a single edge addition or deletion (but not both). To see that under these assumptions the maintenance formula for a φ_1-edge deletion is correct, suppose that there is a suspicious tuple $p(u_1, u_k)$, that is, $S[p, \varphi_1](u_1, u_k) = 1$, but there is a φ_1-path $u_1, \ldots, u_k$ that does not use the deleted φ_1-edge. We need to show that $\mathbf{F}_{st}[p](u_1, u_k)$ has the value 1. Suppose that (a, b) is the φ_1-edge being deleted; because the graph defined by φ_1 is acyclic, there is a $u_i \neq u_k$ that is the last node along path $u_1, \ldots, u_i, u_{i+1}, \ldots, u_k$ from which a is reachable (see Figure 21). Because $p(u_1, u_i)$ and $p(u_{i+1}, u_k)$ both hold, and because u_i cannot be reachable from b (by acyclicity), neither tuple is suspicious; consequently, $T[p, \varphi_1](u_1, u_i) = 1$ and $T[p, \varphi_1](u_{i+1}, u_k) = 1$. Because (u_i, u_{i+1}) is an edge in the new (as well as the old) graph defined by φ_1, we have $\mathbf{F}_{st}[\varphi_1](u_i, u_{i+1}) = 1$, which means that $T[p, \varphi_1](u_i, u_{i+1}) = 1$ as well, yielding $\mathbf{F}_{st}[p](u_1, u_k) = 1$ by Eq. (16).

Figure 22 extends the method for generating relation-maintenance formulas to handle instrumentation relations specified via the **RTC** of a binary formula that defines a directed acyclic graph. Figure 22 makes use of the operator $T[p, \varphi_1](v, v')$ (Eq. (15)), but recasts Eqs. (14) and (16) as finite-difference expressions $\Delta^+_{st}[\psi_p]$ and $\Delta^-_{st}[\psi_p]$, respectively.

φ	$\Delta^+_{st}[\varphi]$	
$p(w_1,\ldots,w_k)$, $p \in \mathcal{I}$	$((\exists\, v\colon \Delta^+_{st}[\varphi_1]) \wedge \neg p)\{w_1,\ldots,w_k\}$	if$\psi_p \equiv \exists\, v\colon \varphi_1$
	$(\exists\, v'_1, v'_2\colon \Delta^+_{st}[\varphi_1](v'_1,v'_2))$ $\wedge \left(\left(\exists\, v'_1, v'_2\colon \begin{array}{c} p(v_1,v'_1) \\ \wedge\, \Delta^+_{st}[\varphi_1](v'_1,v'_2) \\ \wedge\, p(v'_2,v_2) \end{array} \right) \wedge \neg p(v_1,v_2) \right) \{w_1,w_2\}$	if$\psi_p \equiv$ $(\mathbf{RTC}\ v'_1,v'_2\colon \varphi_1)(v_1,v_2)$
	$\Delta^+_{st}[\psi_p]\{w_1,\ldots,w_k\}$	otherwise

φ	$\Delta^-_{st}[\varphi]$	
$p(w_1,\ldots,w_k)$, $p \in \mathcal{I}$	$((\exists\, v\colon \Delta^-_{st}[\varphi_1]) \wedge p)\{w_1,\ldots,w_k\}$	if$\psi_p \equiv \forall\, v\colon \varphi_1$
	$(\exists\, v'_1, v'_2\colon \Delta^-_{st}[\varphi_1](v'_1,v'_2))$ $\wedge \left(\neg \left(\exists\, v'_1, v'_2\colon \begin{array}{c} T[p,\varphi_1](v_1,v'_1) \\ \wedge\, T[p,\varphi_1](v'_1,v'_2) \\ \wedge\, T[p,\varphi_1](v'_2,v_2) \end{array} \right) \wedge p(v_1,v_2) \right) \{w_1,w_2\}$	if$\psi_p \equiv$ $(\mathbf{RTC}\ v'_1,v'_2\colon \varphi_1)(v_1,v_2)$
	$\Delta^-_{st}[\psi_p]\{w_1,\ldots,w_k\}$	otherwise

Fig. 22. Extension of the finite-differencing method from Figure 17 to cover **RTC** formulas, for unit-sized changes to a directed acyclic graph defined by φ_1.

relation p	$\Delta^+_{st}[\psi_p]$
$t_n(v_3,v_4)$	$\Delta^+_{st}[t_n(v_3,v_4)]$ $= (t_n(v_3,v_4) \vee (\exists\, v_1,v_2 : t_n(v_3,v_1) \wedge \Delta^+_{st}[n(v_1,v_2)] \wedge t_n(v_2,v_4))) \wedge \neg t_n(v_3,v_4)$
$r_{n,z}(v)$	$\Delta^+_{st}[r_{n,z}(v)]$ $= (\exists\, v_1 : \Delta^+_{st}[z(v_1) \wedge t_n(v_1,v)]) \wedge \neg r_{n,z}(v)$ $= (\exists\, v_1 : (\Delta^+_{st}[z(v_1)] \wedge \mathbf{F}_{st}[t_n(v_1,v)]) \vee (\mathbf{F}_{st}[z(v_1)] \wedge \Delta^+_{st}[t_n(v_1,v)])) \wedge \neg r_{n,z}(v)$
$c_n(v)$	$\Delta^+_{st}[c_n(v)]$ $= (\exists\, v_1 : \Delta^+_{st}[n(v_1,v) \wedge t_n(v,v_1)]) \wedge \neg c_n(v)$ $= (\exists\, v_1 : (\Delta^+_{st}[n(v_1,v)] \wedge \mathbf{F}_{st}[t_n(v,v_1)]) \vee (\mathbf{F}_{st}[n(v_1,v)] \wedge \Delta^+_{st}[t_n(v,v_1)])) \wedge \neg c_n(v)$

Fig. 23. The formulas obtained via the finite-differencing scheme given in Figures 17 and 22 for the positive changes in the values of the instrumentation relations defined in Figure 20.

Figures 23 and 24 show the formulas obtained via the finite-differencing scheme given in Figures 17 and 22 for positive and negative changes, respectively, for instrumentation relations defined in Figure 20.

6.1.1 *Testing the Unit-Size-Change Assumption.* To know whether this special-case maintenance strategy can be applied, for each statement *st* we need to know at analysis generation time whether the change performed at *st*, to the graph defined by φ_1, always results in a single edge addition or deletion. If in any admissible structure in $\mathcal{S}_2[\mathcal{R}]$ there is a unique satisfying assignment to the two free variables of $\Delta^+_{st}[\varphi_1]$ and no assignment satisfies $\Delta^-_{st}[\varphi_1]$, then the pair $\Delta^+_{st}[\varphi_1]$, $\Delta^-_{st}[\varphi_1]$ defines a change that adds exactly one edge to the graph. Similarly, if in any admissible structure in $\mathcal{S}_2[\mathcal{R}]$ there is a unique satisfying assignment to the two free variables of $\Delta^-_{st}[\varphi_1]$ and no assignment satisfies $\Delta^+_{st}[\varphi_1]$, then the change is a deletion of exactly one edge from the graph.

Because answering (unique-)satisfiability questions in this logic is in general undecidable, we employ a conservative approximation based on a syntactic analysis of logical formulas. The analysis uses a heuristic to determine a set

relation p	$\Delta^-_{st}[p]$
$t_n(v_3,v_4)$	$\Delta^-_{st}[t_n(v_3,v_4)]$ $= (\exists v_1,v_2: T[t_n,n](v_3,v_1) \wedge T[t_n,n](v_1,v_2) \wedge T[t_n,n](v_2,v_4)) \wedge t_n(v_3,v_4)$ $= \left(\exists v_1,v_2: \begin{array}{ll} & (t_n(v_3,v_1) \wedge \neg S[t_n,n](v_3,v_1) \vee \mathbf{F}_{st}[n](v_3,v_1)) \\ \wedge & (t_n(v_1,v_2) \wedge \neg S[t_n,n](v_1,v_2) \vee \mathbf{F}_{st}[n](v_1,v_2)) \\ \wedge & (t_n(v_2,v_4) \wedge \neg S[t_n,n](v_2,v_4) \vee \mathbf{F}_{st}[n](v_2,v_4)) \end{array}\right) \wedge t_n(v_3,v_4)$ $= \left(\exists v_1,v_2: \begin{array}{ll} & (t_n(v_3,v_1) \wedge \neg(\exists v_1',v_2': t_n(v_3,v_1') \wedge \Delta^-_{st}[n](v_1',v_2') \wedge t_n(v_2',v_1)) \vee \mathbf{F}_{st}[n](v_3,v_1)) \\ \wedge & (t_n(v_1,v_2) \wedge \neg(\exists v_1',v_2': t_n(v_1,v_1') \wedge \Delta^-_{st}[n](v_1',v_2') \wedge t_n(v_2',v_2)) \vee \mathbf{F}_{st}[n](v_1,v_2)) \\ \wedge & (t_n(v_2,v_4) \wedge \neg(\exists v_1',v_2': t_n(v_2,v_1') \wedge \Delta^-_{st}[n](v_1',v_2') \wedge t_n(v_2',v_4)) \vee \mathbf{F}_{st}[n](v_2,v_4)) \end{array}\right)$ $\wedge\, t_n(v_3,v_4)$
$r_{n,z}(v)$	$\Delta^-_{st}[r_{n,z}(v)]$ $= \Delta^-_{st}[\exists v_1 : x(v_1) \wedge t_n(v_1,v)]$ $= (\exists v_1 : \Delta^-_{st}[z(v_1) \wedge t_n(v_1,v)]) \wedge \neg(\exists v_1 \mathbf{F}_{st}[z(v_1) \wedge t_n(v_1,v)])$ $= \left\{\begin{array}{l} (\exists v_1 : ((\Delta^-_{st}[z(v_1)] \wedge t_n(v_1,v)) \vee (z(v_1) \wedge \Delta^-_{st}[t_n(v_1,v)]))) \\ \wedge \\ \neg\left(\exists v_1 : \left(\begin{array}{l} (z(v_1) \wedge t_n(v_1,v)) \\ \quad ?\ \neg\Delta^-_{st}[z(v_1) \wedge t_n(v_1,v)] \\ \quad :\ \Delta^+_{st}[z(v_1) \wedge t_n(v_1,v)] \end{array}\right)\right) \end{array}\right.$ $= \left\{\begin{array}{l} (\exists v_1 : ((\Delta^-_{st}[z(v_1)] \wedge t_n(v_1,v)) \vee (z(v_1) \wedge \Delta^-_{st}[t_n(v_1,v)]))) \\ \wedge \\ \neg\left(\exists v_1 : \left(\begin{array}{l} (z(v_1) \wedge t_n(v_1,v)) \\ \quad ?\ \neg((\Delta^-_{st}[z(v_1)] \wedge t_n(v_1,v)) \vee (z(v_1) \wedge \Delta^-_{st}[t_n(v_1,v)])) \\ \quad :\ ((\Delta^+_{st}[z(v_1)] \wedge \mathbf{F}_{st}[t_n(v_1,v)]) \vee (\mathbf{F}_{st}[z(v_1)] \wedge \Delta^+_{st}[t_n(v_1,v)])) \end{array}\right)\right) \end{array}\right.$
$c_n(v)$	$\Delta^-_{st}[c_n(v)]$ $= \Delta^-_{st}[\exists v_1 : n(v_1,v) \wedge t_n(v,v_1)]$ $= (\exists v_1 : \Delta^-_{st}[n(v_1,v) \wedge t_n(v,v_1)]) \wedge \neg\mathbf{F}_{st}[\exists v_1 : n(v_1,v) \wedge t_n(v,v_1)]$ $= (\exists v_1 : (\Delta^-_{st}[n(v_1,v)] \wedge t_n(v,v_1)) \vee (n(v_1,v) \wedge \Delta^-_{st}[t_n(v,v_1)])) \wedge \neg\mathbf{F}_{st}[\exists v_1 : n(v_1,v) \wedge t_n(v,v_1)]$

Fig. 24. The formulas obtained via the finite-differencing scheme given in Figures 17 and 22 for the negative changes in the values of the instrumentation relations defined in Figure 20.

of variables V such that for each admissible structure, the variables in V have a single possible binding in the formula's satisfying assignments. We refer to such variables as *anchored* variables. For instance, if relation q has the attribute "unique," for each admissible structure there is a single possible binding for variable v in any assignment that satisfies $q(v)$; in a formula that contains an occurrence of $q(v)$, v is an anchored variable.

If both free variables of $\Delta^+_{st}[\varphi_1]$ are anchored and $\Delta^-_{st}[\varphi_1] = 0$, then the change adds one edge to the graph defined by φ_1. Similarly, if both free variables of $\Delta^-_{st}[\varphi_1]$ are anchored and $\Delta^+_{st}[\varphi_1] = 0$, then the change removes one edge from the graph. In these cases, the reflexive transitive closure of φ_1 can be updated using the method discussed earlier in the section.

A test for anchored variables. Function *Anchored*, shown in Figure 25, conservatively identifies anchored variables in a formula φ. It is invoked as *Anchored*$(\varphi, \emptyset)$. (In our application, at top-level φ is always either $\Delta^+_{st}[\varphi_1]$ or $\Delta^-_{st}[\varphi_1]$.) *Anchored* uses a handful of patterns to identify anchored variables. For example, if variable v_1 is anchored and binary relation p has the attribute "function,"[9] then v_2 is anchored as well. In essence, negations are handled by pushing the negation deeper into the formula. In a disjunction, an anchored variable must be anchored in both subformulas. The conjunction rule accumulates anchored variables in A by a process of successive approximation, during

[9] For instance, in program analysis applications a relation $n(v_1, v_2)$ that records whether field `n` of v_1 points to v_2 has the "function" attribute.

φ	$\mathit{Anchored}(\varphi, A_0)$
$\mathbf{0}, \mathbf{1}$	A_0
$v_1 = v_2$	$v_1 \in A_0 \rightarrow A_0 \cup \{v_2\} [\!] v_2 \in A_0 \rightarrow A_0 \cup \{v_1\} [\!] A_0$
$p()$	A_0
$p(v)$	$\mathit{unique}(p) \rightarrow A_0 \cup \{v\} [\!] A_0$
$p(v_1, v_2)$	$\mathit{function}(p) \wedge v_1 \in A_0 \rightarrow A_0 \cup \{v_2\}$ $[\!]\ \mathit{invfunction}(p) \wedge v_2 \in A_0 \rightarrow A_0 \cup \{v_1\}$ $[\!]\ A_0$
$\neg\varphi_1$	$\varphi_1 \equiv \neg\varphi_2 \rightarrow \mathit{Anchored}(\varphi_2, A_0)$ $[\!]\ \varphi_1 \equiv \varphi_2 \vee \varphi_3 \rightarrow \mathit{Anchored}(\neg\varphi_2 \wedge \neg\varphi_3, A_0)$ $[\!]\ \varphi_1 \equiv \varphi_2 \wedge \varphi_3 \rightarrow \mathit{Anchored}(\neg\varphi_2 \vee \neg\varphi_3, A_0)$ $[\!]\ \varphi_1 \equiv \forall\, v\colon \varphi_2 \rightarrow \mathit{Anchored}(\exists\, v\colon \neg\varphi_2, A_0)$ $[\!]\ \varphi_1 \equiv \exists\, v\colon \varphi_2 \rightarrow \mathit{Anchored}(\forall\, v\colon \neg\varphi_2, A_0)$ $[\!]\ A_0$
$\varphi_1 \vee \varphi_2$	$\mathit{Anchored}(\varphi_1, A_0) \cap \mathit{Anchored}(\varphi_2, A_0)$
$\varphi_1 \wedge \varphi_2$	$\mu A.(\mathit{Anchored}(\varphi_1, A \cup A_0) \cup \mathit{Anchored}(\varphi_2, A \cup A_0))$
$\exists\, v\colon \varphi_1, \forall\, v\colon \varphi_1$	$(\mathit{Anchored}(\varphi_1, A_0 - \{v\}) - \{v\}) \cup A_0$
$(\mathbf{RTC}\ v_1', v_2'\colon \varphi_1)(v_1, v_2)$	$(\mathit{Anchored}(\varphi_1, A_0 - \{v_1', v_2'\}) - \{v_1', v_2'\}) \cup A_0$

Fig. 25. Function *Anchored* conservatively identifies anchored variables in φ. A_0 contains variables known to be anchored due to the surrounding context.

which variables anchored in the left subformula are used to identify new anchored variables in the right subformula and vice versa; this process is iterated until a fixed point is reached. The rules for $\exists\, v\colon \varphi_1$ and $\forall\, v\colon \varphi_1$ contain recursive calls on *Anchored* with v removed from the second argument (because bound variable v refers to a different occurrence of v from an identically named v in A_0). If v is anchored in φ_1, it needs to be removed before this call returns, to avoid confusion with a v in the outer scope (note the second subtraction of $\{v\}$). Finally, the union of A_0 is performed because v may be in A_0, in which case it has to be included in the answer. $(\mathbf{RTC}\ v_1', v_2'\colon \varphi_1)(v_1, v_2)$ is handled similarly to $\exists\, v\colon \varphi_1$ and $\forall\, v\colon \varphi_1$.

6.2 Transitive Closure Maintenance in Tree-Shaped Graphs

Consider a binary instrumentation relation p, defined by $\psi_p(v_1, v_2) \equiv (\mathbf{RTC}\ v_1', v_2'\colon \varphi_1)(v_1, v_2)$. If the graph defined by φ_1 is not only acyclic but is tree-shaped, it is possible to take advantage of this fact.[10] This fact has no bearing on the maintenance formula that updates the values of relation p after a positive unit-size change $\Delta^+[\varphi_1]$ to the relation φ_1 (see formula (14)). However, it allows the values of p to be updated in a more efficient manner after a negative unit-size change $\Delta^-[\varphi_1]$ to φ_1. In a tree-shaped graph, there exists at most one path between a pair of nodes; if that path goes through the φ_1 edge to be deleted, it should be removed (compare with formula (16)).

$$\mathbf{F}_{st}[p](v_1, v_2) = p(v_1, v_2) \wedge \neg(\exists\, v_1', v_2'\colon p(v_1, v_1') \wedge \Delta^-_{st}[\varphi_1](v_1', v_2') \wedge p(v_2', v_2)) \quad (17)$$

[10] The special-case maintenance strategy that we describe in this subsection also applies only in the case that the change to the graph is a single edge addition or deletion (but not both). We rely on the test described in Section 6.1.1 to ensure that this is the case.

φ	$\Delta^{-}_{st}[\varphi]$	
$p(w_1, \ldots, w_k)$, $p \in \mathcal{I}$	$((\exists\, v\colon \Delta^{-}_{st}[\varphi_1]) \wedge p)\{w_1, \ldots, w_k\}$	if$\psi_p \equiv \forall\, v\colon \varphi_1$
	$(\exists\, v'_1, v'_2\colon p(v_1, v'_1) \wedge \Delta^{-}_{st}[\varphi_1](v'_1, v'_2) \wedge p(v'_2, v_2))\{w_1, w_2\}$	if$\psi_p \equiv$ $(\mathbf{RTC}\ v'_1, v'_2\colon \varphi_1)(v_1, v_2)$
	$\Delta^{-}_{st}[\psi_p]\{w_1, \ldots, w_k\}$	otherwise

Fig. 26. Extension of the finite-differencing method from Figure 17 to cover **RTC** formulas, for unit-sized changes to a tree-shaped graph defined by φ_1. The finite-difference expression $\Delta^{+}_{st}[\psi_p]$ is as defined in Figure 22.

Figure 26 extends the method for generating relation-maintenance formulas to handle instrumentation relations specified via the **RTC** of a binary formula that defines a tree-shaped graph. Figure 26 recasts Eq. (17) as a finite-difference expression $\Delta^{-}_{st}[\psi_p]$.

When comparing the techniques of Section 6.1 for the maintenance of the **RTC** of a binary formula φ_1 with those presented in this subsection, we will refer to the method of Section 6.1 as *acyclic-*φ_1 *maintenance* and the method of this subsection as *tree-shaped-*φ_1 *maintenance*.

6.3 Transitive Closure Maintenance in Deterministic Graphs

A *deterministic graph* is a graph in which every node has outdegree at most one. If the graph defined by φ_1 is deterministic, it is possible to give first-order formulas that maintain reachability information in the graph in response to the addition or deletion of a single φ_1-edge.

The class of deterministic graphs corresponds exactly to the set of possibly-cyclic linked lists. Our solution to the problem of reachability maintenance in possibly-cyclic linked lists can be summarized as follows.

(1) A binary instrumentation relation sfe_n(for *s*panning-*f*orest *e*dge) is introduced to maintain a spanning forest of the (possibly-cyclic) graph defined by the n edges. Thus, we have two types of edges: possibly-cyclic n edges and acyclic sfe_n edges.

(2) We introduce a binary instrumentation relation sfp_n (for *s*panning-*f*orest *p*ath) that captures reachability along the (acyclic) sfe_n edges. sfp_n is the **RTC** of sfe_n, but because sfe_n is acyclic and tree-shaped, sfp_n can be maintained via the techniques described in Section 6.1 or Section 6.2.

(3) We introduce a binary instrumentation relation t_n to capture reachability along n edges. Instead of defining t_n as n^*, as done in Figure 20, we express t_n using first-order logic, based on sfp_n (refer to Figure 32). Thus, t_n can be maintained in terms of sfp_n, via the techniques described in Section 5.

(4) A unary core relation roc_n (for *r*epresentative *o*f the *c*ycle) is introduced to identify a distinguished node of each cycle; the outgoing n edge from a roc_n node is a cycle-breaking edge that is *not* used in the construction of the spanning forest.

In other words, we have a two-level scheme: reachability in the induced, acyclic spanning forest (sfp_n) is maintained via the rules from Section 6.1 or

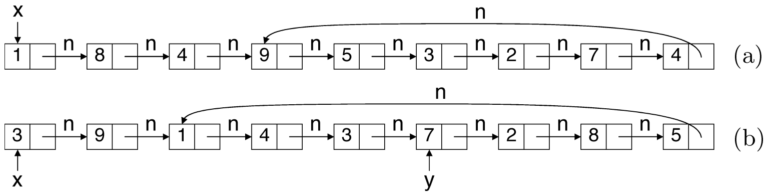


Fig. 27. Possible stores for *panhandle* linked lists. (a) A panhandle list pointed to by x. We will refer to lists of this shape as type-X lists. (b) A panhandle list pointed to by x with y pointing to a node on the cycle. We will refer to lists of this shape as type-XY lists.

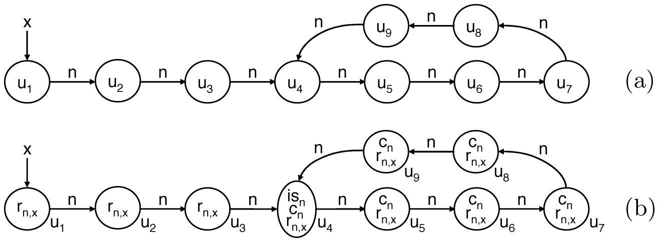


Fig. 28. A logical structure S_{28} that represents the store shown in Figure 27(a) in graphical form: (a) S_{28} with relations from Figure 3; (b) S_{28} with relations from Figures 3 and 6. (Transitive closure relation t_n has been omitted to reduce clutter.)

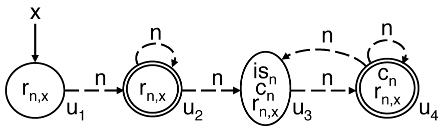


Fig. 29. A 3-valued structure $S^{\#}_{29}$ that is the canonical abstraction of structure S_{28}. In addition to S_{28}, $S^{\#}_{29}$ represents any type-X panhandle list with at least two nodes in the panhandle and at least two nodes in the cycle.

Section 6.2; reachability in the underlying, possibly-cyclic graph (t_n) is then maintained via the rules from Section 5.

6.3.1 *Abstractions of Possibly-Cyclic Linked Lists.* We illustrate our techniques on *panhandle lists*, that is, linked lists that contain a cycle but in which at least the head of the list is not part of the cycle. (The lists shown in Figure 27 are examples of panhandle lists.) Figure 3 gives the definition of a C linked-list datatype, and lists the core relations that would be used to represent the stores manipulated by programs that use type List, such as the stores in Figure 27.

Figure 28 shows two versions of 2-valued structure S_{28}, which represents the store shown in Figure 27(a): Figure 28(a) shows the relations from Figure 3.[11] Figure 28(b) shows the relations from Figure 3, as well as the instrumentation relations from Figure 6.

If all unary relations are abstraction relations (i.e., $\mathcal{A} = \mathcal{R}_1$), the canonical abstraction of 2-valued logical structure S_{28} is $S^{\#}_{29}$, shown in Figure 29, with

[11] We will not show the *dle* relation in the rest of this section because it is not relevant to the problem of reachability maintenance.

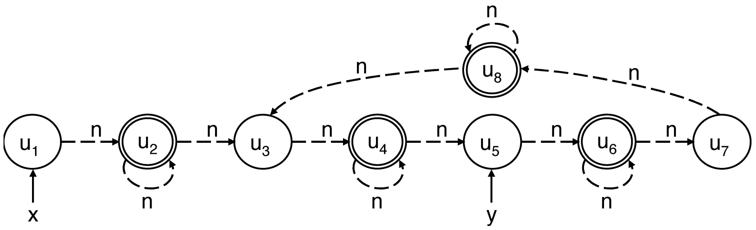


Fig. 30. Logical structure $S^{\#}_{30}$ that represents type-*XY* panhandle lists, such as the store depicted in Figure 27(b). The relations from Figure 6 are omitted to reduce clutter. Their values are as expected for a type-*XY* list: $r_{n,x}$ holds for all nodes, $r_{n,y}$ and c_n hold for all nodes on the cycle, and is_n holds for u_3.

the list nodes corresponding to u_2 and u_3 in S_{28} represented by the summary individual u_2 of $S^{\#}_{29}$, and the list nodes corresponding to u_5, u_6, u_7, u_8, and u_9 in S_{28} represented by the summary individual u_4 of $S^{\#}_{29}$. $S^{\#}_{29}$ represents any type-*X* panhandle list with at least two nodes in the panhandle and at least two nodes in the cycle.

6.3.2 *Reachability Maintenance in Possibly-Cyclic Linked Lists.* Unfortunately, the relations defined in Figures 3 and 6 do not permit precise maintenance of reachability information, such as relation $r_{n,x}$, in possibly-cyclic lists. A difficulty arises when reachability information has to be updated after the deletion of an n edge on a cycle (e.g., as a result of statement `y->n = NULL`). With the relations defined in Figures 3 and 6, such an update requires the reevaluation of a transitive closure formula, which generally results in a drastic loss of precision in the presence of abstraction.

We demonstrate the issue on panhandle lists represented by the abstract structure $S^{\#}_{30}$ shown in Figure 30, that is, lists of type *XY*. (Note that although the store depicted in Figure 27(b) embeds into structure $S^{\#}_{30}$, $S^{\#}_{30}$ is not the canonical abstraction of the store from Figure 27(b); in particular, nodes u_4, u_6, u_7, and u_8 are all indistinguishable according to the instrumentation relations discussed thus far. However, this embedding gives Figure 30 a shape similar to figures that appear later in the section, which will help in illustrating our solution.) Statement `y->n = NULL` has the effect of deleting the `n` edge leaving u_5, thus making the nodes represented by u_6, u_7, and u_8 unreachable from `x`.[12] Note that a first-order-logic formula over the relations of Figures 3 and 6 cannot distinguish the list nodes represented by u_4 from those represented by u_6, u_7, and u_8 in $S^{\#}_{30}$: all of those nodes are reachable from both `x` and `y`, none of those nodes is shared, and all of them lie on a cycle. Our inability to characterize the group of nodes represented by u_4 via a first-order formula requires the maintenance formula for the reachability relation $r_{n,x}$ to recompute some transitive closure information, for example, the transitive closure subformula of the core normal form of the definition of $r_{n,x}$, namely, $n^*(v_1, v)$. However, in the presence of abstraction, reevaluating transitive closure formulas often yields 1/2. For instance, in $S^{\#}_{30}$, formula $n^*(v_1, v)$ evaluates to 1/2 under the assignment

[12] Clearly, all nodes except u_5 also become unreachable from `y`.

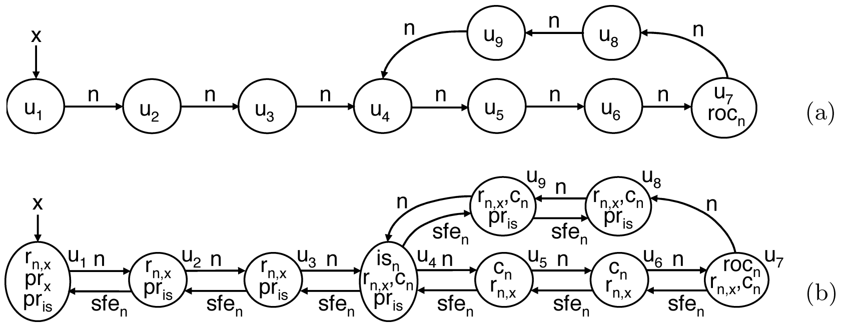


Fig. 31. A logical structure S_{31} that represents the store shown in Figure 27(a) in graphical form: (a) S_{31} with the extended set of core relations. (b) S_{31} with the extended set of core and instrumentation relations (core relations appear in grey). Transitive closure relations sfp_n and t_n have been omitted to reduce clutter. The values of the transitive closure relations can be readily seen from the graphical representation of relations sfe_n and n. For instance, node u_5 is related via the sfp_n relation to itself and all nodes appearing to the left or above it in the pictorial representation.

$[v_1 \mapsto u_1, v \mapsto u_4]$ because of the many 1/2 values of relation n (see the dashed edges connecting u_1 with u_2, for example).

The essence of a solution that enables maintaining reachability relations for possibly-cyclic lists in first-order logic is to find a way to break the symmetry of each cycle. The basic idea for a solution was suggested to us by W. Hesse and N. Immerman. As discussed at the beginning of Section 6.3, it consists of maintaining a spanning-tree representation of a possibly-cyclic list. Reachability in such a representation can be maintained using first-order-logic formulas. Reachability in the actual list can be expressed in first-order logic based on the spanning-tree representation. We now explain our approach and highlight some differences with the approach taken by Hesse [2003].

Our approach relies on the introduction of additional core and instrumentation relations. We extend the set of core relations (Figure 3) with unary relation roc_n, which designates one node on each cycle to be the *r*epresentative *o*f the *c*ycle. (We refer to such a node as a roc_n node.) Relation roc_n is used for tracking a unique *cut edge* on each cycle, which allows the maintenance of a spanning tree. Figure 31(a) shows 2-valued structure S_{31}, which represents the store of Figure 27(a) using the extended set of core relations. Here, we let u_7 be the roc_n node. In general, we simply require that exactly one node on each cycle be designated as a roc_n node. Later in this section we describe how we ensure this.[13]

Figure 32 lists the extended set of instrumentation relations. Note that relation roc_n is not part of the semantics of the language. A natural question is whether $roc_n(v)$ can be defined as an instrumentation relation. For instance,

[13] With the relation-transfer formulas that we use for relation roc_n in this article, the roc_n node for a cycle is the source of the n edge that was inserted to complete the cycle. Note that with this approach, the node that receives the roc_n designation in a given cycle depends on the order of operations that the program performs to construct the cycle.

p	**Intended Meaning**	ψ_p
$is_n(v)$	Do `n` fields of two or more list nodes point to v?	$\exists\, v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$
$sfe_n(v_1, v_2)$	Is there an `n` edge from v_2 to v_1 (assuming that v_2 is not a roc_n node)	$n(v_2, v_1) \wedge \neg roc_n(v_2)$
$sfp_n(v_1, v_2)$	Is v_2 reachable from v_1 along sfe_n edges?	$sfe_n^*(v_1, v_2)$
$t_n(v_1, v_2)$	Is v_2 reachable from v_1 along `n` fields?	$sfp_n(v_2, v_1) \vee \exists\, u, w : \left(\begin{array}{l} sfp_n(u, v_1) \wedge \\ roc_n(u) \wedge n(u, w) \\ \wedge\, sfp_n(v_2, w) \end{array} \right)$
$r_{n,x}(v)$	Is v reachable from pointer variable `x` along `n` fields?	$\exists\, v_1 : x(v_1) \wedge t_n(v_1, v)$
$c_n(v)$	Is v on a directed cycle of `n` fields?	$\exists\, v_1, v_2 : roc_n(v_1) \wedge n(v_1, v_2) \wedge sfp_n(v, v_2)$
$pr_x(v)$	Does v lie on an sfe_n path from `x` (does v *pr*ecede `x` on an `n`-path to a roc_n node)?	$\exists\, v_1 : x(v_1) \wedge sfp_n(v_1, v)$
$pr_{is}(v)$	Does v lie on an sfe_n path from a shared node (does v *pr*ecede a shared node on an `n`-path to a roc_n node)?	$\exists\, v_1 : is_n(v_1) \wedge sfp_n(v_1, v)$

Fig. 32. Defining formulas of instrumentation relations. The sharing relation is_n is defined as in Figure 6. Relations t_n, $r_{n,x}$, and c_n are redefined via first-order-logic formulas in terms of other relations. The exact meaning and purpose of relations sfe_n, sfp_n, pr_x, and pr_{is} will be explained later in the section. Their names stand for *s*panning-*f*orest *e*dge, *s*panning-*f*orest *p*ath, *pr*ecedes x (along a certain path in a cycle), and *pr*ecedes a shared node (along a certain path in a cycle), respectively.

we could try to define it using the following defining formula.

$$c_n(v) \wedge \exists\, v_1 : n(v_1, v) \wedge \neg c_n(v_1) \qquad (18)$$

Formula (18) identifies nodes that lie on a cycle but have a predecessor that does not lie on the cycle. There are three problems with this approach. First, this definition works for panhandle lists but not for cyclic lists without a panhandle. (In general, no other definition can work for cyclic lists without a panhandle because if one existed, it would need to choose one list node among identical-looking nodes that lie on each cycle.) Second, because the cyclicity relation c_n is defined in terms of roc_n (and sfp_n), the definition of roc_n has a circular dependence, which is disallowed. (This circularity cannot be avoided, if we want all reachability relations to benefit from the precise maintenance of one transitive closure relation, here, sfp_n.) The third problem with introducing roc_n as an instrumentation relation is discussed later in the section (see Footnote 15).

We divide our description of the abstraction based on the new set of relations into three parts, which describe (i) how the relations of Figure 32 define *directed* spanning forests, (ii) how we maintain precision on a cycle in the presence of abstraction, and (iii) how we generate maintenance formulas for instrumentation relations *automatically*. The three parts highlight the differences between our approach and that of Hesse.

Defining directed spanning forests. Recall from Section 6.3.2 that the core relations are extended with unary relation roc_n, which designates one node on each cycle to be the *r*epresentative *o*f the *c*ycle. The roc_n nodes can be used to define a (directed) spanning forest of the n edges. Instrumentation relation sfe_n—*sfe* stands for *s*panning-*f*orest *e*dge—is used to maintain the set of edges that forms the spanning forest. In Hesse's work, the spanning-forest edges are directed in the same direction as the n edges; as a result, he maintains a directed spanning forest in which each edge is directed towards the root of a spanning-forest tree. In our work, we define sfe_n to be directed in the direction *opposite* to that of the n edges. The graph defined by the sfe_n relation then defines a directed spanning forest with roc_n nodes as spanning-forest roots, and with each spanning-forest edge directed away from the root of a spanning-forest tree (see Figure 31(b)).

Instrumentation relation sfp_n (*sfp* stands for *s*panning-*f*orest *p*ath) is used to maintain the set of paths in the spanning forest of list nodes. Binary reachability in the actual lists (see relation t_n in Figure 32) can be defined in terms of n, roc_n, and sfp_n using a first-order-logic formula: v_2 is reachable from v_1 if there is a spanning-forest path from v_2 to v_1 or there is a pair of spanning-forest paths, one from the source of a cut edge (a roc_n node) to v_1 and the other from v_2 to the target of the cut edge (the n-successor of the same roc_n node).

Unary reachability relations $r_{n,x}$ and the cyclicity relation c_n can be defined via first-order formulas, as well. We defined $r_{n,x}$ in terms of binary reachability relation t_n. While we could define c_n in terms of t_n as well, we chose another simple definition by observing that a node lies on a cycle if and only if there is a spanning-forest path from it to the target of a cut edge (the n-successor of a roc_n node).

Figure 31(b) shows 2-valued structure S_{31}, which represents the store of Figure 27(a) using the extended set of core and instrumentation relations. The relations pr_x and pr_{is} will be explained shortly.

Preserving node ordering on a cycle in the presence of abstraction. The fact that our techniques need to be applicable in the presence of abstraction introduces a complication that is not present in the setting studied by Hesse. His concern was with the expressibility of certain properties within the confines of a logic with certain syntactic restrictions. Our concern is with the ability to maintain precision in the framework of canonical abstraction.

Unary reachability relations $r_{n,x}$ (one for every program variable x) play a crucial role in the analysis of programs that manipulate acyclic linked lists. In addition to keeping disjoint lists summarized separately, they keep list nodes that have been visited during a traversal summarized separately from nodes that have not been visited: if x is the pointer used to traverse the list, then the nodes that have been visited will have value 0 for relation $r_{n,x}$, while the nodes that have not been visited will have value 1. If a list contains a cycle, then all nodes on the cycle are reachable from the same set of variables, namely, all variables that point to any node in that list. As a result, the instrumentation relations discussed thus far cannot prevent nodes u_4, u_6, and u_8 of $S^{\#}_{30}$ shown in Figure 30 from being summarized together. Thus, assuming that u_7 is the

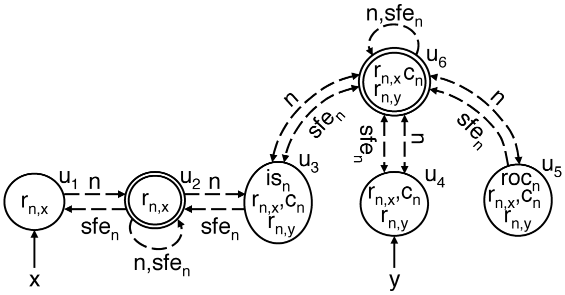

n,sfe$_n$
u$_6$
r$_{n,x}$ c$_n$ r$_{n,y}$
n
sfe$_n$
sfe$_n$
n
n
sfe$_n$
u$_1$
n
r$_{n,x}$
sfe$_n$
u$_2$
n
r$_{n,x}$
sfe$_n$
n,sfe$_n$
u$_3$
is$_n$ r$_{n,x}$,c$_n$ r$_{n,y}$
u$_4$
r$_{n,x}$,c$_n$ r$_{n,y}$
u$_5$
roc$_n$ r$_{n,x}$,c$_n$ r$_{n,y}$
x
y

Fig. 33. A 3-valued structure $S^\#_{33}$ that is the canonical abstraction of structure $S^\#_{30}$ if relations pr_x and pr_{is} are not added to $\mathcal{A}$ and node u_7 of $S^\#_{30}$ (represented by u_5 in $S^\#_{33}$) is the roc_n node.

roc_n node, the canonical abstraction of $S^\#_{30}$ is the 3-valued structure $S^\#_{33}$ shown in Figure 33. The nodes represented by u_4, u_6, and u_8 of $S^\#_{30}$ are represented by the single summary individual u_6 in $S^\#_{33}$. The symmetry hides all information about the order of traversal via pointer variable y. Moreover, the values of the sfp_n relation (not shown in Figure 33) lose precision because ancestors of the shared node in the spanning tree are summarized together with its descendants in the spanning tree.

We break the symmetry of the nodes on a cycle using a general mechanism via unary properties akin to unary reachability relations $r_{n,x}$. In the definitions of relations pr_x of Figure 32, full reachability (relation t_n) has been replaced with spanning-forest reachability (relation sfp_n). The relations pr_x distinguish nodes according to whether or not they are reachable from program variable x along spanning-forest edges. The relation pr_{is} is defined similarly but using instrumentation relation is_n; pr_{is} partitions the nodes of a panhandle list into ancestors and descendants of the shared node in the spanning tree. Figure 34 shows structure $S^\#_{34}$, which is the canonical abstraction of $S^\#_{30}$ of Figure 30, assuming that u_7 is the roc_n node. In $S^\#_{34}$, each of the nodes u_4, u_6, and u_8 has a distinct vector of values for the relations pr_y and pr_{is}, thus breaking the symmetry.[14]

Automatic generation of maintenance formulas for instrumentation relations. In his thesis, Hesse gives hand-specified relation-maintenance formulas for a collection of relations that are used for maintaining a spanning-forest representation of possibly-cyclic linked lists. Instead of specifying relation-maintenance formulas by hand, we rely on finite differencing, as described in previous sections of this article, to generate maintenance formulas for all instrumentation relations. Finite-differencing-generated maintenance formulas have been effective in maintaining all relations defined via first-order-logic formulas, that

[14] Note that in the presence of multiple panhandles, we may be required to introduce finer distinctions to account for the possibility of multiple shared nodes on a cycle. These distinctions could take the form of a family of *is-shared* relations (one for each variable) to capture the panhandle that contributes to sharing. We do not discuss a detailed solution to this problem here, as it will not provide significant further insight.

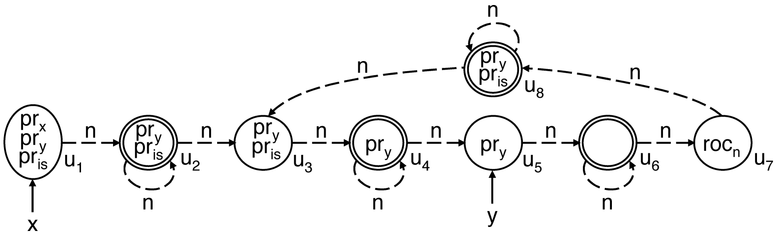


Fig. 34. A 3-valued structure $S^{\#}_{34}$ that is the canonical abstraction of structure $S^{\#}_{30}$ if node u_7 is the roc_n node. $S^{\#}_{34}$ represents panhandle lists of type *XY*, such as the store of Figure 27(b). The only instrumentation relations shown in the figure are pr_x, pr_y, and pr_{is}. As in structure $S^{\#}_{30}$ shown in Figure 30, $r_{n,x}$ holds for all nodes, $r_{n,y}$ and c_n hold for all nodes on the cycle, and is_n holds for u_3.

is, all relations of Figure 32 except sfp_n. Additionally, under certain conditions, finite-differencing-generated maintenance formulas have been effective in maintaining relations defined via the reflexive transitive closure of binary relations. Two conditions are necessary for this technique to be applicable for maintaining relation sfp_n.

Graph-shape condition. The graph defined by sfe_n must be acyclic or tree-shaped.

Unit-size-change condition. Each program statement must only change the graph of n edges by adding a single edge or deleting a single edge (but not both).

The graph-shape condition applies in our setting because the graph defined by sfe_n defines a spanning forest (which is both acyclic and tree-shaped). The unit-size-change condition requires some discussion.

The relation sfe_n is defined in terms of n and roc_n. While we have not yet discussed the relation-update formulas for core relation roc_n, it should be clear that the value of the relation roc_n should only change in response to a change in the value of a node's `n` field. There are two types of statements that change the value of the `n` field and thus may have an effect on the value of the sfe_n relation; namely, statements of the forms `x->n = NULL` and `x->n = y`. The former destroys the n edge leaving the node pointed to by `x`, and the latter creates a new n-connection from the node pointed to by `x` to the node pointed to by `y`. While both of these statements add or remove a single edge of the n relation, it is not necessarily the case that they add or remove a single edge of the sfe_n relation. When interpreted on logical structure $S^{\#}_{34}$ of Figure 34, statement `y->n = NULL` has the effect of deleting the n edge leaving u_5, an action that should result in the deletion of the sfe_n edge entering u_5 (not shown in the figure). However, to preserve the spanning-forest representation, we need to ensure that roc_n holds only for nodes that lie on a cycle, and that sfe_n represents spanning-forest edges. This requires setting the value of roc_n for u_7 to 0 and adding an sfe_n edge from u_8 to u_7. Because, as this example illustrates, a language statement may result in the deletion of one sfe_n edge and the addition of another, neither of the techniques from Section 6.1 and Section 6.2 for maintaining instrumentation relations defined via **RTC** applies.

To work around this problem, the transformers associated with the statements `x->n = NULL` and `x->n = y` each have two phases. In one phase, we apply the part of the transformer that corresponds to the relation n, and update the values of all instrumentation relations. In the other phase, we apply the part of the transformer that corresponds to the relation roc_n, and update the values of all instrumentation relations. As we explain shortly, each phase of the two transformers satisfies the requirement that the change adds a single edge or removes a single edge of the sfe_n relation.[15] Additionally, by paying attention to the order of phases, we ensure that the graph defined by the relation sfe_n remains acyclic and tree-shaped throughout the application of the transformers.

To preserve the graph-shape condition in the case of statement `x->n = NULL`, we apply the part of the transformer that corresponds to the relation n first.

$$\tau_{n,\texttt{x->n = NULL}}(v_1, v_2) = n(v_1, v_2) \wedge \neg x(v_1) \tag{19}$$

Unless `x` points to a roc_n node (or `x->n` is `NULL`), this phase results in the deletion of the sfe_n edge that enters the node pointed to by `x`. In the second phase, we apply the part of the transformer that corresponds to the relation roc_n.

$$\tau_{roc_n,\texttt{x->n = NULL}}(v) = roc_n(v) \wedge \exists\, v_1 : n(v, v_1) \wedge sfp_n(v, v_1) \tag{20}$$

This phase sets the roc_n property of the source n_s of a cut edge to 0, if there is no longer a spanning-forest path from n_s to the target n_t of the same cut edge. When this happens and `x` does not point to n_s, that is, the cut edge is not being deleted, this phase results in the addition of an sfe_n edge from n_t to n_s.

To preserve the graph-shape condition in the case of statement `x->n = y`, we apply the part of the transformer that corresponds to the relation roc_n first.

$$\tau_{roc_n,\texttt{x->n = y}}(v) = roc_n(v) \vee (x(v) \wedge \exists\, v_1 : y(v_1) \wedge sfp_n(v, v_1)) \tag{21}$$

If there is a spanning-forest path from node n_x, pointed to by `x`, to node n_y, pointed to by `y`, the statement creates a new cycle in the data structure. The update of formula (21) sets the roc_n property of n_x to 1, thus making n_x the source of a new cut edge and n_y the target of the cut edge. Because there was no n edge from n_x to n_y prior to the execution of this statement,[16] this phase results in no change to the sfe_n relation. In the second phase, we apply the part of the transformer that corresponds to the relation n.

$$\tau_{n,\texttt{x->n = y}}(v_1, v_2) = n(v_1, v_2) \vee (x(v_1) \wedge y(v_2)) \tag{22}$$

Unless the node pointed to by `x` became a roc_n node in the first phase, this phase results in the addition of an sfe_n edge from n_y to n_x.

The break-up of the transformers corresponding to statements `x->n = NULL` and `x->n = y` into two phases, as described before, ensures that the sfe_n relation

[15] The third problem with defining roc_n as an instrumentation relation (alluded to earlier in the section) is that we would lose the ability to apply the two parts of a transformer separately: the change in the values of n would immediately trigger a change in the values of roc_n. The resulting transformer would not be able to satisfy the unit-size-change condition.

[16] By normalizing procedures to include a statement of the form `x->n = NULL` prior to a statement of the form `x->n = y`, we ensure that `x->n` is always `NULL` prior to the latter assignment.

remains acyclic and tree-shaped throughout the analysis (the graph-shape condition) and that the change to the sfe_n relation that results from each phase is a unit-size change (the unit-size-change condition).[17] Thus, it is sound to maintain sfp_n ($= sfe_n^*$) via the techniques described in either Section 6.1 or Section 6.2. Additionally, it is also sound to maintain the remaining instrumentation relations via the techniques of Section 5 because the remaining relations are defined by first-order-logic formulas. Soundness guarantees that the stored values of instrumentation relations agree with the relations' defining formulas throughout the analysis. However, the stored values may not agree with the relations' *intended meanings*. For instance, if the `n`-transfer phase of the transformer for statement `x->n = NULL` removes a noncut `n` edge on a cycle, the sfe_n relation will temporarily not span the entire list. However, we do not query the results of abstract interpretation *in between* the phases of a two-phase transformer. Thus, the stored values of instrumentation relations agree with the relations' intended meanings, as well as their defining formulas, at all points in the program's control-flow graph.

Optimized maintenance of relation sfp_n. By demonstrating that the acyclicity and unit-size-change conditions hold for relation sfe_n, we were able to rely on the techniques of Section 6.1 to maintain the relation sfp_n. Note, however, that the definition of sfe_n ensures that the graph defined by sfe_n is not only acyclic but is tree-shaped. This fact has no bearing on the maintenance formula that updates the values of relation sfp_n after a positive unit-size change $\Delta^+[sfe_n]$ to sfe_n (see formula (14)). However, it allows the values of sfp_n to be updated in a more efficient manner after a negative unit-size change $\Delta^-[sfe_n]$ to sfe_n. In a tree-shaped graph, there exists at most one path between a pair of nodes; if that path goes through the sfe_n edge to be deleted, the corresponding sfp_n edge should be removed (compare with formula (16)).

$$\begin{aligned}\mathbf{F}_{st}[sfp_n](v_1, v_2) = \;& sfp_n(v_1, v_2)\\ \wedge\;& \neg(\exists\, v_1', v_2' \colon sfp_n(v_1, v_1') \wedge \Delta^-[sfe_n](v_1', v_2') \wedge sfp_n(v_2', v_2))\end{aligned} \tag{23}$$

7. EXPERIMENTAL EVALUATION

To evaluate the techniques presented in Section 5 and Section 6, we extended TVLA version 2 to generate relation-maintenance formulas, and applied it to a test suite of five existing analysis specifications, involving twenty-four programs, along with a variety of different abstractions and properties to check (see Figure 35). The experiment was designed to determine what penalty is incurred when the relation-maintenance formulas generated by our finite-differencing-based algorithm are used in place of hand-crafted relation-maintenance formulas. In the experiment, we used the set of hand-crafted relation-maintenance formulas that had been built up during several years of experience with TVLA. The idea was that if the penalty is low for the programs, abstractions, and program properties in the test suite, that would be evidence that the penalty

[17] The test described in Section 6.1.1 confirms that the unit-size-change condition holds for each phase.

Category	Test Program	# of non-identity maintenance formulas				Performance				
		schemas			# inst.	Analysis Time (sec.)			% increase	
						Ref.	FD		FD	
		total	TC	non-TC			acyc.	tree	acyc.	tree
SLL Shape Analysis	Search	2	0	2	2	0.30	0.30	0.31	1.10	1.90
	GetLast	3	0	3	4	0.31	0.32	0.32	2.23	2.22
	DeleteAll	11	2	9	15	0.30	0.32	0.30	4.97	-0.13
	Reverse	12	2	10	16	0.43	0.49	0.44	12.69	1.99
	Create	11	2	9	21	0.28	0.31	0.28	9.61	-0.60
	Delete	12	2	10	39	1.13	2.13	1.23	87.90	7.76
	Merge	11	2	9	64	1.77	3.67	1.96	107.27	10.42
	Insert	12	2	10	72	1.19	2.03	1.31	70.43	9.67
DLL Shape Analysis	Append	15	2	13	50	1.76	1.78	1.77	1.13	0.57
	Delete	16	2	14	74	8.35	8.78	8.38	5.15	0.36
	Splice	15	2	13	96	1.06	1.69	1.10	59.70	3.79
Binary Tree Shape Analysis	InsertSorted	13	2	11	43	1.25	1.28	1.28	1.97	1.54
	Lindstrom	10	2	8	43	40.44	82.29	41.48	103.47	2.57
	DSW	10	2	8	52	101.30	180.20	109.51	77.89	8.15
	DeleteSorted	13	2	11	554	75.26	409.31	97.71	443.85	29.69
SLL Sorting	ReverseSorted	18	2	16	23	0.47	0.54	0.49	13.05	2.58
	BubbleSort	18	2	16	80	5.74	8.91	6.42	55.32	11.77
	BubbleSortBug	18	2	16	80	5.41	7.61	6.01	40.75	11.14
	InsertSortBug2	18	2	16	87	5.19	17.57	6.09	238.55	17.04
	InsertSort	18	2	16	88	5.65	18.55	6.66	228.26	17.95
	InsertSortBug1	18	2	16	88	18.94	32.93	20.25	73.84	7.27
	MergeSorted	18	2	16	91	2.26	4.22	2.53	86.35	11.46
Information Flow	Good Flow	12	2	10	66	13.59	23.28	15.37	71.30	13.59
	Bad Flow	12	2	10	86	78.05	180.85	94.92	131.70	21.79

Fig. 35. Results from using hand-crafted vs. automatically generated maintenance formulas for instrumentation relations.

will be low for other examples; and that one can afford to forgo the effort of hand-crafting maintenance formulas for other analysis examples.

The test programs consisted of various operations on acyclic singly-linked lists, doubly-linked lists, binary trees, and binary-search trees, plus several sorting programs [Lev-Ami et al. 2000]. The system was used to verify a variety of properties of the test programs. For instance, Reverse, an in-situ list reversal program, must preserve list properties and lose no elements; InsertSorted and DeleteSorted must preserve binary-search-tree properties; InsertSort must return a sorted list; Good Flow must not allow high-security input data to flow to a low-security output channel. (Loginov et al. discuss the verification of stronger properties, such as the *partial correctness* of several of the algorithms [Loginov et al. 2005, 2007; Loginov 2006].)

Lindstrom and DSW are two variants of Deutsch-Schorr-Waite, a constant-space tree-traversal algorithm that uses destructive pointer rotation. For Lindstrom and DSW, we verified that the algorithms have no unsafe pointer operations or memory leaks, and that the data structure produced at the end is, in fact, a binary tree. (Loginov et al. [2006] discuss the verification of the *total correctness* of Deutsch-Schorr-Waite—i.e., that the binary tree produced at the end is identical to the input tree and that the algorithm terminates.)

A few of the programs contained bugs: for instance, InsertSortBug2 is an insert-sort program that ignores the first element of the list; BubbleBug is a bubble-sort program with an incorrect condition for swapping elements, which causes an infinite loop if the input list contains duplicate data values. (See Lev-Ami et al. [2000], Dor et al. [2000], and Lev-Ami and Sagiv [2000] for more details.)

In TVLA, the operational semantics of a programming language is defined by specifying, for each kind of statement, an *action schema* to be used on outgoing CFG edges. Action schemas are instantiated according to a program's statement instances to create the CFG. For each combination of action schema and instrumentation relation, a *maintenance formula schema* must be provided. The number of nonidentity maintenance formula schemas is reported in columns 3–5 of Figure 35 (grouped under the header "schemas"). In columns 4–5, they are broken down into those whose defining formula contains an occurrence of **RTC** (under the header "TC"), and those that do not (under the header "non-TC"). Relation-maintenance formulas produced by finite differencing are generally larger than the hand-crafted ones. Because this affects analysis time, the number of instances of nonidentity maintenance formula schemas is a meaningful size measure for our experiments. These numbers appear in column 6 (under the header "# inst."). The number of instances of nonidentity schemas for `DeleteSorted` is high because `DeleteSorted` includes three inline expansions of the routine that finds the tree node that takes the place of the deleted node.[18]

The data structures manipulated by all programs in our test suite are acyclic and tree-shaped, thus acyclic reachability maintenance (i.e., the technique described in Section 6.1), as well as tree-shaped reachability maintenance (i.e., the technique described in Section 6.2), apply for the maintenance of reachability relations. In the absence of hand-crafted maintenance formulas for reachability relations in possibly-cyclic linked lists, we could not extend our experiments to cover the technique of Section 6.3. Instead, we validated that technique as part of the verification of properties of `Reverse` when applied to possibly-cyclic linked lists (see Loginov et al. [2007]).

For each program in the test suite, we first ran the analysis using hand-crafted maintenance formulas, to obtain a reference answer in which CFG nodes were annotated with their final sets of logical structures. We then ran the analysis using automatically generated maintenance formulas with acyclic reachability maintenance (Section 6.1) and compared the result against the reference answer. For all 24 test programs, the analysis using automatically generated formulas yielded answers identical to the reference answers, thus establishing the same properties. Finally, we ran the analysis using automatically generated maintenance formulas with tree-shaped reachability maintenance (Section 6.2) and compared the result against the reference answer. Again, for all 24 test programs, the analysis using automatically generated formulas yielded answers identical to the reference answers, thus establishing the same properties.

Columns 7–11 show performance data, which were collected on a 3GHz PC with 3.7GB of RAM running CentOS 4 Linux. The column labeled "Ref." gives the reference times (i.e., when the hand-crafted maintenance formulas are

[18] Work on interprocedural shape analysis provides a solution that does not require inline-expanded programs [Rinetzky and Sagiv 2001; Rinetzky et al. 2005; Jeannet et al. 2004, 2010].

used). Columns labeled "acyc." give the data for the analyses that used automatically generated maintenance formulas with acyclic reachability maintenance. Columns labeled "tree" give the data for the analyses that used automatically generated maintenance formulas with tree-shaped reachability maintenance. (As stated earlier, for each test program the use of hand-crafted maintenance formulas, acyclic reachability maintenance formulas, and tree-shaped reachability maintenance formulas all established identical properties.) In each case, five runs were made; the longest and shortest times were discarded from each set, and the remaining three averaged. The geometric mean of the slowdowns when using the automatically generated formulas with acyclic reachability maintenance was approximately 60%, with a median of 55%, mainly due to the fact that the automatically generated formulas are larger than the hand-crafted ones. The maximum slowdown was 444%. The highest slowdowns occurred in analyses of programs that involved deletions of edges in a data structure's graph.

Because the edge-deletion maintenance formulas produced by the tree-shaped reachability-maintenance technique are much smaller than those that are produced by acyclic reachability maintenance, our expectation was that the use of tree-shaped reachability-maintenance formulas would cause a much smaller slowdown. This expectation was confirmed: the geometric mean of the slowdowns when using the automatically generated formulas with tree-shaped reachability maintenance was approximately 8%, with a median of 7%. The maximum slowdown was 30%.[19] A few analyses were actually faster with the automatically generated formulas; these speedups are either due to random variation or are accidental benefits of subformula orderings that are advantageous for short-circuit evaluation.

These results are encouraging. At least for abstractions of several common data structures, they suggest that the algorithm for generating relation-maintenance formulas from Section 5 and Section 6 is capable of automatically generating formulas that (i) are as precise as the hand-crafted ones, and (ii) have a tolerable effect on runtime performance.

The extended version of TVLA also uncovered several bugs in the hand-crafted formulas. A maintenance formula of the form $\mu_{p,st}(v_1, \ldots, v_k) = p(v_1, \ldots, v_k)$ is called an *identity relation-maintenance formula*. For each identity relation-maintenance formula in the hand-crafted specification, we checked that (after simplification) the corresponding generated relation-maintenance formula was also an identity formula. Each inconsistency turned out to be an error in the hand-crafted specification. We also found one instance of an incorrect nonidentity hand-crafted maintenance formula. (The measurements reported in Figure 35 are based on corrected hand-crafted specifications.)

8. RELATED WORK

Our work addresses an instance of the following fundamental challenge in applying abstract interpretation:

[19] We expect that some simple optimizations, such as caching the results from evaluating subformulas, could reduce the slowdown further.

> Given the concrete semantics for a language and a desired abstraction, how does one create sound abstract transformers?

In our context, a desired abstraction is specified by defining the set of instrumentation relations to use. The question is how to obtain sound relation-maintenance formulas for use in the abstract transformer. A weakness of the original formulation of TVLA [Sagiv et al. 2002; Lev-Ami and Sagiv 2000] was that the user needed to define relation-maintenance formulas by hand to specify how each structure transformer affects each instrumentation relation. Past criticisms of TVLA based on this deficiency [Ball et al. 2001; Møller and Schwartzbach 2001] are no longer valid, at least for analyses that can be specified using formulas that define acyclic relations (and also for some classes of formulas that define cyclic relations). With the algorithm presented in Section 5 and Section 6, the user's responsibility is merely to write the ψ_p formulas that define the set of instrumentation relations; appropriate relation-maintenance formulas are then created automatically.

For certain situations [Graf and Saïdi 1997; Reps et al. 2004b; Yorsh et al. 2004], it is known how to create best abstract transformers in the sense of Cousot and Cousot [1979]; that is, the abstract transformers created are the most precise transformers that are possible, given the abstraction that is in use. Graf and Saïdi [1997] showed that theorem provers can be used to generate best abstract transformers for abstract domains that are fixed, finite, Cartesian products of Boolean values. (The use of such domains is known as *predicate abstraction*; predicate abstraction is also used in SLAM [Ball et al. 2001] and other systems [Das et al. 1999].) Reps et al. [2004b] gave a method, which applies to a broader class of abstract domains than predicate abstraction domains, for computing the effect of applying the best transformer. Yorsh et al. [2004] gave a related method for abstract domains based on canonical abstraction (i.e., for shape analysis). Both of these methods provide a way to apply a best transformer, but do not produce an explicit representation of the abstract transformer. Moreover, they each involve a sequence of calls on an appropriate decision procedure (or semidecision procedure). In the case of shape analysis, the method of Yorsh et al. [2004] is generally much more expensive than the algorithm described in Section 5 and Section 6.

In contrast, the abstract transformers created using the algorithm described in Section 5 and Section 6 are not best transformers; however, the algorithm uses only very simple, linear-time, recursive tree-traversal procedures, whereas the theorem provers used in predicate abstraction are not even guaranteed to terminate. Moreover, our setting makes available much richer abstract domains than the ones offered by predicate abstraction, and experience to date has been that very little precision is lost (using only *good* abstract transformers) once the right instrumentation relations have been identified.

Other work on automatically creating abstract transformers includes the following:

—Methods based on *semantic reinterpretation* [Mycroft and Jones 1985, 1986; Nielson 1989; Malmkjær 1993; Lim and Reps 2008] factor the concrete semantics of a language into two parts: (i) a client specification, and (ii) a semantic core. The interface to the core consists of certain base types, function types, and operators, and the client is expressed in terms of this interface. Such an organization permits the core to be reinterpreted to produce an alternative semantics for the subject language.

Semantic reinterpretation is convenient in that soundness of the *entire* abstract semantics can be established via purely *local* soundness arguments for each of the reinterpreted operators. However, semantic reinterpretation has a purely local (and hence myopic) view of the behavior of a transformer, and hence can lead to abstract transformers that lose a substantial amount of precision. In contrast, the algorithm presented in Section 5 and Section 6 is generally able to retain an appropriate amount of precision because the finite-differencing approach to obtaining an abstract transformer for a concrete transformer τ aims to reuse as much information as possible from the prestate structure, and thereby avoids formula reevaluation operations for tuples of a relation whose values cannot be changed by τ.

—Scherpelz et al. [2007] developed a method for creating abstract transformers for use with parameterized predicate abstraction [Cousot 2003]. It uses Weakest Liberal Precondition (WLP) followed by heuristics that approximate how combinations of prestate relations imply the WLP of a poststate relation with respect to a transformer τ. Generating the abstract transformer for a (nullary) instrumentation relation p in vocabulary $\mathcal{R}$ involves two steps (where p is defined by the nullary formula $\psi_p()$):

(1) Create the formula $\varphi = \mathrm{WLP}(\tau, \psi_p())$.

(2) Find a Boolean combination $\mu_{p,\tau}[\mathcal{R}]$ of the prestate relations such that if $\mu_{p,\tau}[\mathcal{R}]$ holds in the prestate then φ must also hold in the prestate (and hence p must hold in the poststate).

The abstract transformer is a function that sets the value of p in the poststate according to whether $\mu_{p,\tau}[\mathcal{R}]$ holds in the prestate.

Because WLP performs substitution on $\psi_p()$, the formula created by step (1) is related to the naïve relation-maintenance formula defined in Eq. (5) of Section 3. Step (2) applies several heuristics to φ to produce one or more strengthenings of φ; step (2) returns the disjunction of the strengthened variants of φ. In contrast, the algorithm presented in Section 5 and Section 6 does not operate by trying to strengthen the naïve relation-maintenance formula; instead, it uses a systematic approach, based on finite differencing of p's defining formula $\psi_p()$, to identify how τ changes p's value. Moreover, our method is not restricted to nullary instrumentation relations: it applies to relations of arbitrary arity.

A special case of canonical abstraction occurs when no abstraction relations are used at all, in which case all individuals of a logical structure are collapsed to a single individual. When this is done, in almost all structures the only useful information remaining resides in the nullary core and instrumentation relations. Predicate abstraction can be seen as going one step further, retaining only the nullary instrumentation relations (and *no* abstracted core relations). However, to be able to evaluate a “Future” formula (as defined in

> Eqs. (11) and (12) of Section 5) such as $\mathbf{F}_\tau[p] \overset{\text{def}}{=} p ? \neg\Delta^-_\tau[p] : \Delta^+_\tau[p]$, one generally needs the prestate abstract structure to hold (abstracted) core relations. From this standpoint, our method and that of Scherpelz et al. [2007] are incomparable; they have different goals, and neither can be said to subsume the other.

Paige and Koenig [1982] studied how finite-differencing transformations of applicative set-former expressions could be exploited to optimize loops in very high-level languages, such as SETL. Liu et al. used related program transformation methods in the setting of a functional programming language to derive incremental algorithms for various problems from the specifications of exhaustive algorithms [Liu and Teitelbaum 1995; Liu et al. 1996]. In their work, the goal is to maintain the value of a function $F(x)$ as the input x undergoes small changes. The methods described in Section 5 and Section 6 address a similar kind of incremental computation problem, except that the language in which the exhaustive and incremental versions of the problem are expressed is first-order logic with reflexive transitive closure.

The finite-differencing operators defined in Section 5 and Section 6 are most closely related to a number of previous papers on logic and databases: finite-difference operators for the propositional case were studied by Akers [1959] and Sharir [1982]. Previous work on incrementally maintaining materialized views in databases [Gupta and Mumick 1999], "first-order incremental evaluation schemes (FOIES)" [Dong and Su 1995], and "dynamic descriptive complexity" [Patnaik and Immerman 1997] has also addressed the problem of maintaining one or more auxiliary relations after new tuples are inserted into or deleted from the base relations. In databases, view maintenance is solely an optimization; the correct information can always be obtained by reevaluating the formula. In the abstract-interpretation context, where abstraction has been performed, this is no longer true: reevaluating a formula in the local (3-valued) state can lead to a drastic loss of precision. Thus, one aspect that sets our work apart from previous work is the goal of developing a finite-differencing transformation suitable for use when abstraction has been performed.

In Section 6.3.2, we compared our work to that of Hesse [2003], which is closest in spirit to our techniques for maintaining reachability information in possibly-cyclic linked lists. Next, we discuss a few approaches that bear resemblance to ours in that they attempt to translate or simulate a data structure that cannot be handled by some core techniques into one that can.

The idea of using spanning-tree representations for specifying or reasoning about data structures that are "close to trees" is not new. Klarlund and Schwartzbach introduced *graph types*, which can be used to specify some common non-tree-shaped data structures in terms of a spanning-tree backbone and regular expressions that specify where nonbackbone edges occur within the backbone [Klarlund and Schwartzbach 1993]. Examples of data structures that can be specified by graph types are doubly-linked lists and threaded trees. A panhandle list cannot be specified by a graph type because in a graph type the location of each nonbackbone edge has to be defined in terms of the backbone using a regular expression, and a regular expression cannot be used to specify the existence of a backedge to *some* node that occurs earlier in the list. In the PALE project [Møller and Schwartzbach 2001], which incorporates work on graph types, automated reasoning about programs that manipulate data structures specified as graph types can be carried out using a decision procedure for monadic second-order logic. Unfortunately, the decision procedure has nonelementary complexity. An advantage of our approach over that of PALE is that we do not rely on the use of a decision procedure.

Immerman et al. [2004] presented *structure simulation*, a technique that broadens the applicability of decision procedures to a larger class of data structures. Under certain conditions, it allows data structures that cannot be reasoned about using decidable logics to be translated into data structures that can, with the translation expressed as a first-order-logic formula. Unlike graph types, structure simulation is capable of specifying panhandle lists. However, this technique shares a limitation of graph types because it relies on decision procedures for automated reasoning about programs.

Manevich et al. [2005] specified abstractions (in canonical-abstraction and predicate-abstraction forms) for showing safety properties of programs that manipulate possibly-cyclic linked lists. By maintaining reachability within list segments that are not *interrupted* by nodes that are shared or pointed to by a variable, they are able to break the symmetry of a cycle. The definition of several key instrumentation relations in that work makes use of transitive closure formulas that cannot be handled precisely by finite differencing. As a result, a drawback of that work is the need to define some relation-maintenance formulas by hand. Another drawback is the difficulty of reasoning about reachability (in a list) from a program variable (see reachability relations $r_{n,x}$ of Figure 32). Because in Manevich et al. [2005] reachability in a list has to be expressed in terms of reachability over a sequence of uninterrupted segments, a formula that expresses the reachability of node v from program variable x in a list has to enumerate all permutations of other program variables that may act as interruptions on a path from x to v in the list.

A number of past approaches to the analysis of programs that manipulate linked lists relied on first-order axiomatizations of reachability information. All of these approaches involved the use of first-order-logic decision procedures. While our approach does not have this limitation, it is instructive to compare our work with those approaches that included mechanisms for breaking the symmetry on a cycle. Nelson [1983] defined a set of first-order axioms that describe the ternary reachability relation $r_n(u, v, w)$, which has the meaning: w is reachable from u along n edges without encountering v. The use of this relation alone is not sufficient in our setting because in the presence of abstraction we require unary distinctions (such as the relations pr_x and pr_{is} of Figure 32) to break the symmetry. Additionally, the maintenance of ternary relations is more expensive than the maintenance of binary relations. Lahiri and Qadeer [2006] specify a collection of first-order axioms that are sufficient to verify properties of procedures that perform a single change to a cyclic list, for example, the removal of an element. They also verify properties of in-situ list reversal, albeit under the assumption that the input list is acyclic. In a recent publication, we describe a case study in which we use the techniques developed in Section 6.3 to verify the *total correctness* (partial correctness and termination) of `Reverse` when applied to any linked list, including cyclic and panhandle lists [Loginov et al. 2007]. Lahiri and Qadeer break the symmetry of cycles in a similar fashion to how it is done by Manevich et al. [2005]: the *blocking cells* of Lahiri and Qadeer [2006] are a subset of the interruptions of Manevich et al. [2005]. The blocking cells include only the set of *head variables*, program variables that act as heads of lists used in the program. This set has to be maintained carefully by the user to (i) satisfy the system's definition of acceptable (*well-founded*) lists, (ii) allow the system to verify useful postconditions, and (iii) avoid falling prey to the difficulty (which arises in the work of Manevich et al. [2005]) of expressing reachability in the list. The current mechanism of Lahiri and Qadeer [2006] is insufficient for reasoning about panhandle lists because the set of blocking cells does not include shared nodes. This limitation can be partially addressed by generalizing the set of blocking cells to mimic interruptions of Manevich et al. [2005] more faithfully. However, this may make it more difficult to satisfy points (ii) and (iii) given before. As in our work, Lahiri and Qadeer rely on the insight that reachability information can be maintained in first-order logic. They use a collection of manually specified update formulas that define how their relations are affected by the statements of the language and the (user-inserted) statements that manage the set of head variables.

Distefano et al. [2006] presented a shape analysis algorithm for singly-linked lists based on separation logic. As shown by Reynolds [2002], one of the advantages of separation logic is that, for some programs, it can be used to specify invariants in an intuitive way. In most simple list-manipulating programs, the invariants are much more succinct than ones produced by TVLA [Yorsh et al. 2007].[20] Also, as shown by Ishtiaq and O'Hearn [2001], the separating conjunction of separation logic presents a simpler way to express postconditions than is possible in the framework of Sagiv et al. [2002] (see also Jeannet et al. [2010]).

However, the framework of Sagiv et al. provides several benefits compared with existing domains based on separation logic ("separation domains").

- The framework of Sagiv et al. handles arbitrary imperative programs, and can prove arbitrary properties that can be expressed in first-order logic with reflexive transitive closure (including numeric properties [Gopan et al. 2004, 2005]). In contrast, existing separation domains concentrate on proving memory safety and preservation of data structure invariants in linked lists.
- The framework of Sagiv et al. can be applied to arbitrary data structures, whereas existing separation domains can be applied to data structures without sharing (e.g., acyclic singly-linked lists and binary trees) and data

[20] Yorsh et al. [2007] presented a method that, given a 3-valued structure $S^{\#}$, creates a formula $\varphi[S^{\#}]$ that is satisfied by exactly the set of structures that $S^{\#}$ represents. For most 3-valued structures, such formulas are quite complicated.

structures with sharing that can be defined explicitly (e.g., binary trees with parent pointers), but not to data structures that exhibit more complex sharing patterns (e.g., DAGs).

—Canonical abstraction is based on an intuitive abstraction principle. Moreover, results can be rendered in a natural way (see Section 2.2). Both features make it easier to understand where information is lost, compared with results obtained from tools that use separation domains.

—For programs that manipulate data structures beyond singly-linked lists, and for proving properties beyond memory safety, it is challenging to guarantee that an analysis using a separation domain terminates: the abstract domain is infinite, and a widening operation is not known that guarantees that the analysis will terminate in a finite number of iterations. This situation is the case in all existing separation domains, for example, the one described by Berdine et al. [2007]. In contrast, abstract domains based on canonical abstraction are finite, and hence termination is guaranteed.

The techniques used to create abstract transformers for canonical-abstraction domains and separation domains are quite different. In our approach, the next state is determined using three primitives (see Section 2.2.2 and Sagiv et al. [2002]): (i) partial concretization (or partial model enumeration) via the *focus* operation, (ii) formula evaluation, using the finite-differencing-based formulas created by the techniques presented in Section 5 and Section 6, and (iii) very lightweight logical reasoning via the *coerce* operation. Analyses based on separation logic use specialized decision procedures and formula normalization procedures. Existing tools based on separation domains spend a lot of time in formula normalization and in entailment checks.[21] In part, this is because the separating conjunction does not distribute over ordinary logical conjunction, but the primary reason for the high cost is that the separating conjunction involves an implicit second-order quantification. Consequently, the overall cost of an analysis specified using the framework of Sagiv et al. can be comparable to, or even lower, than that of an analysis based on a separation domain. (For a comparison, see Bogudlov et al. [2007b].)

9. CONCLUSIONS

This article addresses a fundamental challenge that arises in abstract interpretation:

> Given the concrete semantics for a language and a desired abstraction, how does one create the associated abstract transformers?

[21] In the instantiation of the framework of Sagiv et al. that we extended in this work (namely, TVLA version 2), the *coerce* operation dominates the cost of analysis. In our experiments, execution of *coerce* accounted for 56%–98% of analysis time, with a mean of 72%. Aggregated over all test cases, execution of *coerce* accounted for 85% of the total analysis time. Bogudlov et al. studied the cost of key operations in TVLA. They achieved substantial speedups (as much as 50-fold) due to a number of optimizations of the *coerce* operation. The reader is referred to their tool paper and the accompanying technical report for more details [Bogudlov et al. 2007a, 2007b].

This challenge arises in program analysis problems in which the semantics of statements is expressed using logical formulas that describe changes to core relation values. When instrumentation relations have been introduced to refine an abstraction, the challenge is to reflect the changes in core relation values in the values of the instrumentation relations. The algorithm presented in this article provides a way to create formulas that maintain correct values for the instrumentation relations, and thereby provides a way to generate (completely automatically) the part of the transformers of an abstract semantics that deals with instrumentation relations.

The work described in this article opened the way for TVLA to be extended to support *automatic abstraction refinement* [Loginov et al. 2005; Loginov 2006]. The idea is to start the analyzer with a crude abstraction, and use the results of analysis runs that fail to establish a definite answer (about whether the property of interest does or does not hold) as feedback about how the abstraction should be refined.

Abstraction refinement had previously been used in the model-checking community [Kurshan 1994; Clarke et al. 2000; Ball and Rajamani 2001] however, finding an analog of this that was suitable for TVLA was a challenging problem because canonical abstraction is considerably more sophisticated than the abstractions used by the model-checking community: in particular, predicate abstraction [Graf and Saïdi 1997] can be viewed as the degenerate case of canonical abstraction in which only nullary relations are retained [Reps et al. 2004a, Section 4]. Because of this difference, we found that we had to use mechanisms that were completely different from those used in tools such as SLAM [Ball and Rajamani 2001], BLAST [Henzinger et al. 2002], and Magic [Chaki et al. 2003]. Our solution involved using inductive logic programming to discover an appropriate set of instrumentation relations that refine the abstraction in use [Loginov et al. 2005; Loginov 2006]. Finite differencing is a crucial enabling technique in this approach because it provides the ability to create relation-maintenance formulas automatically after refinement has been performed.

Finally, although the work described in the article was motivated by a problem that arose in work on static analysis based on 3-valued logic, any method in which systems are described as evolving (2-valued or 3-valued) logical structures (e.g., Alloy [Jackson 2006] or Abstract State Machines [Boerger and Staerk 2003]) may be able to benefit from these techniques.

ACKNOWLEDGMENTS

We are grateful to W. Hesse, N. Immerman, T. Lev-Ami, and R. Wilhelm for their comments and suggestions concerning this work. R. Manevich provided invaluable help with TVLA.

REFERENCES

AKERS, JR., S. 1959. On a theory of Boolean functions. *J. Soc. Indust. Appl. Math. 7*, 4, 487–498.

BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the Conference on Programming Language Design and Implementation*. 203–213.

BALL, T. AND RAJAMANI, S. 2001. Automatically validating temporal safety properties of interfaces. In *Proceedings of the SPIN Workshop*. 103–122.

BERDINE, J., CALCAGNO, C., COOK, B., DISTEFANO, D., O'HEARN, P., WIES, T., AND YANG, H. 2007. Shape analysis for composite data structures. *In Proceedings of the Conference on Computer-Aided Verification*. 178–192.

BOERGER, E. AND STAERK, R. 2003. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer.

BOGUDLOV, I., LEV-AMI, T., REPS, T., AND SAGIV, M. 2007a. Revamping TVLA: Making parametric shape analysis competitive. Tech. rep. TR-2007-01-01, Tel-Aviv University, Tel-Aviv, Israel.

BOGUDLOV, I., LEV-AMI, T., REPS, T., AND SAGIV, M. 2007b. Revamping TVLA: Making parametric shape analysis competitive (tool paper). In *Proceedings of the Conference on Computer-Aided Verification*. 221–225.

CHAKI, S., CLARKE, E., GROCE, A., JHA, S., AND VEITH, H. 2003. Modular verification of software components in C. In *Proceedings of the International Conference on Software Engineering*. 385–395.

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.

COUSOT, P. 2003. Verification by abstract interpretation. In *Verification: Theory and Practice*. 243–268.

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 *Proceedings of the Conference on Principles of Programming Languages*. 238–252.

COUSOT, P. AND COUSOT, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the Conference on Principles of Programming Languages*. 269–282.

DAS, S., DILL, D., AND PARK, S. 1999. Experience with predicate abstraction. In *Proceedings of the Conference on Computer-Aided Verification*. 160–171.

DISTEFANO, D., O'HEARN, P., AND YANG, H. 2006. A local shape analysis based on separation logic. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 287–302.

DONG, G. AND SU, J. 1995. Incremental and decremental evaluation of transitive closure by first-order queries. *Inform. Comput. 120*, 101–106.

DONG, G. AND SU, J. 2000. Incremental maintenance of recursive views using relational calculus/SQL. *SIGMOD Rec. 29*, 1, 44–51.

DOR, N., RODEH, M., AND SAGIV, M. 2000. Checking cleanness in linked lists. In *Proceedings of the Static Analysis Symposium*. 115–134.

GOLDSTINE, H. 1977. *A History of Numerical Analysis*. Springer.

GOPAN, D., DIMAIO, F., DOR, N., REPS, T., AND SAGIV, M. 2004. Numeric domains with summarized dimensions. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 512–529.

GOPAN, D., REPS, T., AND SAGIV, M. 2005. A framework for numeric analysis of array operations. In *Proceedings of the Conference on Principles of Programming Languages*. 338–350.

GRAF, S. AND SAÏDI, H. 1997. Construction of abstract state graphs with PVS. In *Proceedings of the Conference on Computer-Aided Verification*. 72–83.

GUPTA, A. AND MUMICK, I., EDS. 1999. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, Cambridge, MA.

HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy abstraction. In *Proceedings of the Conference on Principles of Programming Languages*. 58–70.

HESSE, W. 2003. Dynamic computational complexity. Ph.D. thesis, Department of Computer Science, University of Massachusetts, Amherst, MA.

IMMERMAN, N., RABINOVICH, A., REPS, T., SAGIV, M., AND YORSH, G. 2004. Verification via structure simulation. In *Proceedings of the Conference on Computer-Aided Verification*. 281–294.

ISHTIAQ, S. AND O'HEARN, P. 2001. BI as an assertion language for mutable data structures. In *Proceedings of the Conference on Principles of Programming Languages*. 14–26.

JACKSON, D. 2006. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA.

JEANNET, B., LOGINOV, A., REPS, T., AND SAGIV, M. 2004. A relational approach to interprocedural shape analysis. In *Proceedings of the Static Analysis Symposium*. 246–264.

JEANNET, B., LOGINOV, A., REPS, T., AND SAGIV, M. 2010. A relational approach to interprocedural shape analysis. *Trans. Program. Lang. Syst. 32*, 2.

KLARLUND, N. AND SCHWARTZBACH, M. 1993. Graph types. In *Proceedings of the Conference on Principles of Programming Languages*. 196–205.

KURSHAN, R. 1994. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press.

LAHIRI, S. AND QADEER, S. 2006. Verifying properties of well-founded linked lists. In *Proceedings of the Conference on Principles of Programming Languages*. 115–126.

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.

LIM, J. AND REPS, T. 2008. A system for generating static analyzers for machine instructions. In *Proceedings of the Conference on Compiler Construction*. 36–52.

LIU, Y., STOLLER, S., AND TEITELBAUM, T. 1996. Discovering auxiliary information for incremental computation. In *Proceedings of the Conference on Principles of Programming Languages*. 157–170.

LIU, Y. AND TEITELBAUM, T. 1995. Systematic derivation of incremental programs. *Sci. Comput. Program. 24*, 2, 1–39.

LOGINOV, A. 2006. Refinement-based program verification via three-valued-logic analysis. Ph.D. thesis, Tech. rep. 1574. Computer Science Department, University of Wisconsin, Madison, WI.

LOGINOV, A., REPS, T., AND SAGIV, M. 2005. Abstraction refinement via inductive learning. In *Proceedings of the Conference on Computer-Aided Verification*. 519–533.

LOGINOV, A., REPS, T., AND SAGIV, M. 2006. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *Proceedings of the Static Analysis Symposium*. 261–279.

LOGINOV, A., REPS, T., AND SAGIV, M. 2007. Refinement-based verification for possibly-cyclic lists. In *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*. 247–272.

MALMKJÆR, K. 1993. Abstract interpretation of partial-evaluation algorithms. Ph.D. thesis, Department of Computer and Information Science, Kansas State University, Manhattan, KS.

MANEVICH, R., YAHAV, E., RAMALINGAM, G., AND SAGIV, M. 2005. Predicate abstraction and canonical abstraction for singly-linked lists. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation*. 181–198.

MCMILLAN, K. 1999. Verification of infinite state systems by compositional model checking. In *Proceedings of the Conference on Correct Hardware Design and Verification Methods (CHARME'99)*. 219–234.

MICHIE, D. 1968. Memo functions and machine learning. *Nature 218*, 19–22.

MØLLER, A. AND SCHWARTZBACH, M. 2001. The pointer assertion logic engine. In *Proceedings of the Conference on Programming Language Design and Implementation*. 221–231.

MYCROFT, A. AND JONES, N. 1985. A relational framework for abstract interpretation. In *Programs as Data Objects*. 156–171.

MYCROFT, A. AND JONES, N. 1986. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the Conference on Principles of Programming Languages*. 296–306.

NELSON, G. 1983. Verifying reachability invariants of linked structures. In *Proceedings of the Conference on Principles of Programming Languages*. 38–47.

NIELSON, F. 1989. Two-level semantics and abstract interpretation. *Theor. Comput. Sci. 69*, 2, 117–242.

PAIGE, R. AND KOENIG, S. 1982. Finite differencing of computable expressions. *Trans. Program. Lang. Syst. 4*, 3, 402–454.

PATNAIK, S. AND IMMERMAN, N. 1997. Dyn-FO: A parallel, dynamic complexity class. *J. Comput. Syst. Sci. 55*, 2, 199–209.

REPS, T., LOGINOV, A., AND SAGIV, M. 2002. Semantic minimization of 3-valued propositional formulae. In *Proceedings of the Conference on Logic in Computer Science*. 40–54.

REPS, T., SAGIV, M., AND LOGINOV, A. 2003. Finite differencing of logical formulas for static analysis. In *Proceedings of the European Symposium on Programming*. 380–398.

REPS, T., SAGIV, M., AND WILHELM, R. 2004a. Static program analysis via 3-valued logic. In *Proceedings of the Conference on Computer-Aided Verification*. 15–30.

REPS, T., SAGIV, M., AND YORSH, G. 2004b. Symbolic implementation of the best transformer. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation*. 252–266.

REYNOLDS, J. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Conference on Logic in Computer Science*. 55–74.

RINETZKY, N., BAUER, J., REPS, T., SAGIV, M., AND WILHELM, R. 2005. A semantics for procedure local heaps and its abstractions. In *Proceedings of the Conference on Principles of Programming Languages*. 296–309.

RINETZKY, N. AND SAGIV, M. 2001. Interprocedural shape analysis for recursive programs. In *Proceedings of the Conference on Compiler Construction*. 133–149.

SAGIV, M., REPS, T., AND WILHELM, R. 2002. Parametric shape analysis via 3-valued logic. *Trans. Program. Lang. Syst. 24*, 3, 217–298.

SCHERPELZ, E., LERNER, S., AND CHAMBERS, C. 2007. Automatic inference of optimizer flow functions from semantic meanings. In *Proceedings of the Conference on Programming Language Design and Implementation*. 135–145.

SHARIR, M. 1982. Some observations concerning formal differentiation of set theoretic expressions. *Trans. Program. Lang. Syst. 4*, 2, 196–225.

TVLA. TVLA system. www.cs.tau.ac.il/~tvla/.

VAN FRAASSEN, B. 1966. Singular terms, truth-value gaps, and free logic. *J. Phil. 63*, 17, 481–495.

YORSH, G., REPS, T., AND SAGIV, M. 2004. Symbolically computing most-precise abstract operations for shape analysis. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 530–545.

YORSH, G., REPS, T., SAGIV, M., AND WILHELM, R. 2007. Logical characterizations of heap abstractions. *Trans. Comput. Logic 8*, 1.

Received March 2009; revised August 2009; accepted December 2009