# Finite Differencing of Logical Formulas for Static Analysis[⋆]

Thomas Reps[1], Mooly Sagiv[2], and Alexey Loginov[1]

[1] Comp. Sci. Dept., University of Wisconsin; {reps,alexey}@cs.wisc.edu
[2] School of Comp. Sci., Tel-Aviv University; msagiv@post.tau.ac.il

**Abstract.** This paper concerns mechanisms for maintaining the value of an instrumentation predicate (a.k.a. *derived predicate* or *view*), defined via a logical formula over core predicates, in response to changes in the values of the core predicates. It presents an algorithm for transforming the instrumentation predicate's defining formula into a *predicate-maintenance formula* that captures what the instrumentation predicate's new value should be.

This technique applies to program-analysis problems in which the semantics of statements is expressed using logical formulas that describe changes to core-predicate values, and provides a way to reflect those changes in the values of the instrumentation predicates.

## 1 Introduction

This paper addresses a fundamental challenge in applying abstract interpretation, namely,

> 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-predicate values. When instrumentation predicates (defined via logical formulas over the core predicates) have been introduced to refine an abstraction, the challenge is to reflect the changes in core-predicate values in the values of the instrumentation predicates [8,5, 14,18,3]. The algorithm presented in this paper provides a way to create formulas that maintain correct values for the instrumentation predicates, and thereby provides a way to generate, completely automatically, the part of the transfer functions of an abstract semantics that deals with instrumentation predicates. The algorithm runs in time linear in the size of the instrumentation predicate's defining formula.

This research was motivated by our work on static analysis based on 3-valued logic [18]; 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, we consider two related logics: 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

[⋆] Supported by ONR contract N00014-01-1-0796 and by the A. von Humboldt Foundation.

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. 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 [11,1].

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, some concrete individuals "lose their identity" when they are grouped together with other individuals in one summary individual. Moreover, 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. This provides a partial answer to the fundamental challenge posed above: formulas that define the concrete semantics when interpreted in 2-valued logic define a sound abstract semantics when interpreted in 3-valued logic [18].

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 predicates*, which record auxiliary information in a logical structure. They provide a mechanism for the user to fine-tune an abstraction: an instrumentation predicate, which is defined by a logical formula over the core predicate symbols, captures a property that an individual memory cell may or may not possess. In general, adding additional instrumentation predicates refines the abstraction, defining a more precise analysis that is prepared to track finer distinctions among stores. This allows more properties of the program's stores to be identified.

From the standpoint of the concrete semantics, instrumentation predicates represent cached information that could always be recomputed by reevaluating the instrumentation predicate'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 predicates, an abstract-interpretation algorithm must obtain their values in some other way.

This problem, the *instrumentation-predicate-maintenance problem*, will be solved by incremental computation. The new value that instrumentation predicate $p$ should have after a transition via abstract state transformer $\tau$ from state $\sigma$ to $\sigma'$ will be computed incrementally from the known value of $p$ in $\sigma$.

The contributions of the work reported in this paper include the following:

– We give an algorithm for the predicate-maintenance problem; it creates a predicate-maintenance formula by applying a finite-differencing transformation to $p$'s defining formula. The algorithm runs in time linear in the size of the defining formula.
– We present experimental evidence that our technique is an effective one, at least for the analysis of programs that manipulate acyclic singly-linked lists, doubly-linked lists, and binary trees, and for certain sorting programs. In particular, the predicate-maintenance formulas produced automatically using our approach are as effective for maintaining precision as the best available hand-crafted ones.
– This work is related to the view-maintenance problem in databases. Compared with that work, the novelty is the ability to create predicate-maintenance formulas that are suitable for use when abstraction has been performed.

The remainder of the paper is organized as follows: Sect. 2 introduces terminology and notation. Sect. 3 defines the predicate-maintenance problem. Sect. 4 presents a method for generating maintenance formulas for instrumentation predicates. Sect. 5 discusses extensions to handle instrumentation predicates that use transitive closure. Sect. 6 presents experimental results. Sect. 7 discusses related work.

## 2 Background

**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 1.** A *formula* over the *vocabulary* $\mathcal{P} = \{eq, p_1, \ldots, p_n\}$ is defined by

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

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 $\varphi_1$'s free-variable set is $V$, we require $v_1, v_2 \notin V$. The free variables of $\varphi$ 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 predicate $p$, $p^*(v_1, v_2) \stackrel{\text{def}}{=} (\mathbf{RTC}\; v_1', v_2' \colon p(v_1', v_2'))(v_1, v_2)$. We also use a C-like syntax for conditional expressions: $\varphi_1 \;?\; \varphi_2 : \varphi_3$.[1] The order 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.

[1] In 2-valued logic, one can think of $\varphi_1 \;?\; \varphi_2 : \varphi_3$ 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 $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$.

**Definition 2.** A *2-valued interpretation* over $\mathcal{P}$ is a *2-valued logical structure* $S = \langle U^S, \iota^S \rangle$, where $U^S$ is a set of *individuals* and $\iota^S$ maps each predicate symbol $p$ of arity $k$ 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 $\varphi$, we say that $Z$ is *complete* for $\varphi$. (We generally assume that every assignment that arises in connection with the discussion of some formula $\varphi$ is complete for $\varphi$.)

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

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

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

$S$ and $Z$ *satisfy* $\varphi$ if $[\![\varphi]\!]_2^S(Z) = 1$. The set of 2-valued structures is denoted by 2-STRUCT[$\mathcal{P}$].

**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 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 $l_2 = 1/2$. 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 $\sqsubseteq$.

**Definition 4.** A *3-valued interpretation* over $\mathcal{P}$ is a *3-valued logical structure* $S = \langle U^S, \iota^S \rangle$, where $U^S$ is a set of individuals and $\iota^S$ maps each predicate symbol $p$ of arity $k$ 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 $\varphi$, denoted by $[\![\varphi]\!]_3^S(Z)$, yields a truth value in $\{0, 1/2, 1\}$. The meaning of $\varphi$ is defined exactly as in Defn. 2, but interpreted over $\{0, 1/2, 1\}$. $S$ and $Z$ *potentially satisfy* $\varphi$ if $[\![\varphi]\!]_3^S(Z) \sqsupseteq 1$. The set of 3-valued structures is denoted by 3-STRUCT[$\mathcal{P}$].

Defn. 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 abstract-interpretation context, a summary individual is an abstract individual, and can represent more than one concrete individual.

Because $\varphi_1 ? \varphi_2 : \varphi_3$ is treated as a shorthand for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$ in 3-valued logic, the value of $1/2 \; ? \; V_1 : V_2$ equals $V_1 \sqcup V_2$.

**Definition 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 predicate symbol $p \in \mathcal{P}$ of arity $k$ 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 $\varphi$ is a conservative approximation of the information extracted from $S$ via $\varphi$. 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 1. (Embedding Theorem [18, 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 $\varphi$ and complete assignment $Z$ for $\varphi$, $[\![\varphi]\!]_3^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S'}(f \circ Z)$.*

**Program Analysis Via 3-Valued Logic.** The remainder of this section summarizes the program-analysis framework described in [18]. Stores are encoded as logical structures in terms of a fixed collection of *core predicates*, $\mathcal{C}$. Core predicates are part of the underlying semantics of the language to be analyzed; they record atomic properties of stores. For instance, Tab. 1 gives the definition of a C linked-list datatype, and lists the predicates that would be used to represent the stores manipulated by programs that use type `List`. (The core predicates are fixed for a given language; in general, different languages require different collections of core predicates.)

**Table 1.** (a) Declaration of a linked-list datatype in C. (b) Core predicates used for representing the stores manipulated by programs that use type `List`.

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

(a)

| Predicate | Intended Meaning |
|---|---|
| $eq(v_1, v_2)$ | Do $v_1$ and $v_2$ denote the same memory cell? |
| $q(v)$ | Does pointer variable q point to memory cell $v$? |
| $n(v_1, v_2)$ | Does the n field of $v_1$ point to $v_2$? |

(b)

Often only a restricted class of structures is used to encode stores; to exclude structures that cannot represent admissible stores, integrity constraints can be imposed. For instance, in program-analysis applications, a predicate like $q(v)$ of Tab. 1 captures whether pointer variable q points to memory cell $v$; $q$ would be given the attribute "unique", which imposes the integrity constraint that $q$ can hold for at most one individual in any structure.

A concrete operational semantics is defined by specifying, for each kind of statement $st$ in the programming language, a structure transformer for each outgoing control-flow graph (CFG) edge $e = (st, st')$. A structure transformer is specified by providing a collection of *predicate-transfer formulas*, $\tau_{c,st}$, one for each core predicate $c$. These define how the core predicates of a logical structure $S$ that arises at $st$ is transformed by $e$ to create a logical structure $S'$ at $st'$.

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 $S^\natural$ to those of a 3-valued structure $S$. The Embedding Theorem ensures that every piece of information extracted from $S$ by evaluating a formula $\varphi$ is a conservative approximation ($\sqsupseteq$) of the information extracted from $S^\natural$ by evaluating $\varphi$.

The finiteness of the abstract domain is assured by *canonical abstraction*, under which 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 predicates. This mechanism ensures that each 3-valued structure is no larger than some fixed size, known *a priori*.

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 predicates* to record derived properties, and (ii) varying which of the unary core and unary instrumentation predicates are used as the set of abstraction predicates. The set of instrumentation predicates is denoted by $\mathcal{I}$. Each arity-$k$ predicate symbol $p \in \mathcal{I}$ is defined by an *instrumentation-predicate definition formula* $\psi_p(v_1, \ldots, v_k)$. Instrumentation predicates may appear in the defining formulas of other instrumentation predicates as long as there are no circular dependences. Instrumentation predicates 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. This is particularly important when analyzing a program in which two pointers are advanced along disjoint sublists.

For each kind of statement in the programming language, the abstract semantics is again defined by a collection of formulas: the *same* predicate-transfer formula that defines the concrete semantics, in the case of a core predicate, and, in the case of an instrumentation predicate $p$, by a *predicate-maintenance formula* $\mu_{p,st}$.[2]

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 [18]. (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.)

Fig. 1 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.

The following graphical notation is used for depicting 3-valued logical structures:

– Individuals are represented by circles with names inside.
– A summary individual $u$ has $eq(u, u) = 1/2$, and is represented by a double circle.

[2] In [18], predicate-transfer formulas and predicate-maintenance formulas are both called "predicate-update formulas". Here we use separate terms so that we can refer easily to predicate-maintenance formulas, which are the main subject of the paper.

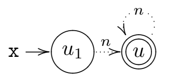
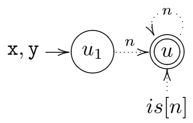
| | |
|---|---|
| Structure before | **unary preds.** $\begin{array}{c\|ccc} \textbf{indiv.} & x & y & is[n] \\ \hline u_1 & 1 & 0 & 0 \\ u & 0 & 0 & 0 \end{array}$ **binary preds.** $\begin{array}{c\|cc} n & u_1 & u \\ \hline u_1 & 0 & 1/2 \\ u & 0 & 1/2 \end{array}$ $\begin{array}{c\|cc} eq & u_1 & u \\ \hline u_1 & 1 & 0 \\ u & 0 & 1/2 \end{array}$  |
| Statement | `y = x` |
| Predicate-transfer formulas | $\tau_{x,\mathtt{y=x}}(v) = x(v)$<br>$\tau_{y,\mathtt{y=x}}(v) = x(v)$<br>$\tau_{n,\mathtt{y=x}}(v_1, v_2) = n(v_1, v_2)$ |
| Predicate-maintenance formula | $\mu_{is[n],\mathtt{y=x}}(v) = \exists\, v_1, v_2\colon n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$ |
| Structure after | **unary preds.** $\begin{array}{c\|ccc} \textbf{indiv.} & x & y & is[n] \\ \hline u_1 & 1 & 1 & 0 \\ u & 0 & 0 & 1/2 \end{array}$ **binary preds.** $\begin{array}{c\|cc} n & u_1 & u \\ \hline u_1 & 0 & 1/2 \\ u & 0 & 1/2 \end{array}$ $\begin{array}{c\|cc} eq & u_1 & u \\ \hline u_1 & 1 & 0 \\ u & 0 & 1/2 \end{array}$  |

**Fig. 1.** The predicate-transfer formulas for $x$, $y$, and $n$ express a transformation on logical structures that corresponds to the semantics of $\mathtt{y} = \mathtt{x}$. (The predicate $is[n]$ is discussed in Ex. 1.)

- A unary predicate $p$ is represented by a solid arrow from $p$ to each individual $u$ for which $\iota(p)(u) = 1$, and by the absence of a $p$-arrow to each node $u'$ for which $\iota(p)(u') = 0$. (If $\iota(p) = 0$ for all individuals, the predicate name $p$ is not shown.)
- A binary predicate $q$ is represented by a solid arrow labeled $q$ between each pair of individuals $u_i$ and $u_j$ for which $\iota(q)(u_i, u_j) = 1$, and by the absence of a $q$-arrow between pairs $u'_i$ and $u'_j$ for which $\iota(q)(u'_i, u'_j) = 0$.
- Unary and binary predicates with value $1/2$ are represented by dotted arrows.

## 3 The Problem: Maintaining Instrumentation Predicates

The execution of a statement $st$ transforms a logical structure $S$, which represents a store that arises just before $st$, into a new structure $S'$, which represents the corresponding store just after $st$ executes. The structure that consists of just the core predicates of $S'$ is called a *proto-structure*, denoted by $S'_{proto}$. The creation of $S'_{proto}$ from $S$, denoted by $S'_{proto} := [\![st]\!]_3(S)$, can be expressed as

$$\text{for each } c \in \mathcal{C} \text{ and } u_1, \ldots, u_k \in U^S,$$
$$\iota^{S'_{proto}}(c)(u_1, \ldots, u_k) := [\![\tau_{c,st}(v_1, \ldots, v_k)]\!]_3^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]). \quad (1)$$

In general, if we compare the various predicates of $S'_{proto}$ with those of $S$, some tuples will have been added and others will have been deleted.

We now come to the crux of the matter: Suppose that $\psi_p$ defines instrumentation predicate $p$; how should the static-analysis engine obtain the value of $p$ in $S'$?

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

$$\text{for each } u_1, \dots, u_k \in U^S,$$
$$\iota^{S'}(p)(u_1, \dots, u_k) := [\![\psi_p(v_1, \dots, v_k)]\!]_3^{S'_{proto}}([v_1 \mapsto u_1, \dots, v_k \mapsto u_k]). \quad (2)$$

Thus, in principle it is possible to maintain the values of instrumentation predicates via Eqn. (2). In practice, however, this approach does not work very well. As observed elsewhere [18], when working in 3-valued logic, it is usually possible to retain more precision by defining a special *instrumentation-predicate maintenance formula*, $\mu_{p,st}(v_1, \dots, v_k)$, and evaluating $\mu_{p,st}(v_1, \dots, v_k)$ in structure $S$:

$$\text{for each } u_1, \dots, u_k \in U^S,$$
$$\iota^{S'}(p)(u_1, \dots, u_k) := [\![\mu_{p,st}(v_1, \dots, v_k)]\!]_3^{S}([v_1 \mapsto u_1, \dots, v_k \mapsto u_k]). \quad (3)$$

The advantage of the predicate-maintenance approach is that the results of program analysis can be more accurate. In 3-valued logic, when $\mu_{p,st}$ is defined appropriately, the predicate-maintenance strategy can generate a definite value (0 or 1) when the evaluation of $\psi_p$ 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 6.** Suppose that $p$ is an instrumentation predicate defined by formula $\psi_p$. Predicate-maintenance formula $\mu_{p,st}$ *maintains* $p$ *correctly for statement* $st$ if, for all $S \in$ 2-STRUCT[$\mathcal{P}$] and all $Z$, $[\![\mu_{p,st}]\!]_2^S(Z) = [\![\psi_p]\!]_2^{[\![st]\!]_2(S)}(Z)$.

For an instrumentation predicate in core normal form, it is always possible to provide a predicate-maintenance formula that satisfies Defn. 6 by defining $\mu_{p,st}$ as

$$\mu_{p,st} \stackrel{\text{def}}{=} \psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}], \quad (4)$$

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

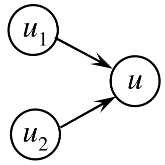


**Fig. 2.** Store in which $u$ is shared; i.e., $is[n](u) = 1$.

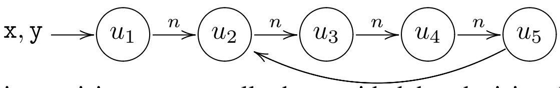
The formula $\mu_{p,st}$ defined in Eqn. (4) maintains $p$ correctly for statement $st$ because, by the 2-valued version of Eqn. (1), $[\![\tau_{c,st}]\!]_2^S(Z) = [\![c]\!]_2^{S'_{proto}}(Z)$; consequently, when $\mu_{p,st}$ of Eqn. (4) is evaluated in structure $S$, the use of $\tau_{c,st}$ in place of $c$ is equivalent to using the value of $c$ when $\psi_p$ is evaluated in $S'_{proto}$; i.e., for all $Z$, $[\![\psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}]]\!]_2^S(Z) = [\![\psi_p]\!]_2^{S'_{proto}}(Z)$. However—and this is precisely the drawback of using Eqn. (4) to obtain the $\mu_{p,st}$—the steps of evaluating $[\![\psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}]]\!]_2^S(Z)$ *mimic exactly* those of evaluating $[\![\psi_p]\!]_2^{S'_{proto}}(Z)$. Consequently, when we pass to 3-valued logic, for all $Z$, $[\![\psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}]]\!]_3^S(Z)$ yields exactly the same value as $[\![\psi_p]\!]_3^{S'_{proto}}(Z)$ (i.e., as evaluating Eqn. (2)). Thus, although $\mu_{p,st}$ that satisfy Defn. 6 can be obtained automatically via Eqn. (4), this approach does not provide a satisfactory solution to the predicate-maintenance problem.

*Example 1.* Eqn. (5) shows the defining formula for the instrumentation predicate $is[n]$ ("is-shared using n fields"),

$$is[n](v) \stackrel{\text{def}}{=} \exists\, v_1, v_2 \colon n(v_1, v) \wedge n(v_2, v) \wedge v_1 {\neq} v_2, \tag{5}$$

which captures whether a memory cell is pointed to by two or more pointer fields of memory cells, e.g., see Fig. 2.

Fig. 1 illustrates how execution of the statement $\mathtt{y} = \mathtt{x}$ causes the value of $is[n]$ to lose precision when its predicate-maintenance formula is created according to Eqn. (4). 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 $\mathtt{y} = \mathtt{x}$ does not change the value of core predicate $n$, $\tau_{n,\mathtt{y}=\mathtt{x}}(v_1, v_2)$ is $n(v_1, v_2)$, and hence the formula $\mu_{is[n],\mathtt{y}=\mathtt{x}}(v)$ created according to Eqn. (4) is $\exists\, v_1, v_2 \colon n(v_1, v) \wedge n(v_2, v) \wedge v_1 {\neq} v_2$. As shown in Fig. 1, the structure created using this maintenance formula is not as precise as we would like. In particular, $is(u) = 1/2$, which means that $u$ can represent a shared cell. Thus, the final 3-valued structure also represents certain cyclic linked lists, such as



$$\begin{tikzcd} \mathtt{x}, \mathtt{y} \arrow[r] & u_1 \arrow[r, "n"] & u_2 \arrow[r, "n"] & u_3 \arrow[r, "n"] & u_4 \arrow[r, "n"] & u_5 \arrow[llll, bend left] \end{tikzcd}$$

This sort of imprecision can usually be avoided by devising better predicate-maintenance formulas. For instance, when $\mu_{is[n],\mathtt{y}=\mathtt{x}}(v)$ is defined to be the formula $is[n](v)$—meaning that $\mathtt{y} = \mathtt{x}$ does not change the value of $is[n](v)$—the imprecision illustrated in Fig. 1 is avoided (see Fig. 3). Hand-crafted predicate-maintenance formulas for a variety of instrumentation predicates are given in [18,11,1]; however, those formulas were created by *ad hoc* methods.

**Structure before**

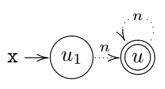
unary preds.

| indiv. | $x$ | $y$ | $is[n]$ |
|---|---|---|---|
| $u_1$ | 1 | 0 | 0 |
| $u$ | 0 | 0 | 0 |

binary preds.

| $n$ | $u_1$ | $u$ |
|---|---|---|
| $u_1$ | 0 | 1/2 |
| $u$ | 0 | 1/2 |

| $eq$ | $u_1$ | $u$ |
|---|---|---|
| $u_1$ | 1 | 0 |
| $u$ | 0 | 1/2 |



| | |
|---|---|
| Statement | `y = x` |
| Predicate-transfer formulas | $\tau_{x,\mathtt{y}=\mathtt{x}}(v) = x(v)$ <br> $\tau_{y,\mathtt{y}=\mathtt{x}}(v) = x(v)$ <br> $\tau_{n,\mathtt{y}=\mathtt{x}}(v_1, v_2) = n(v_1, v_2)$ |
| Predicate-maintenance formula | $\mu_{is[n],\mathtt{y}=\mathtt{x}}(v) = is[n](v)$ |

**Structure after**

unary preds.

| indiv. | $x$ | $y$ | $is[n]$ |
|---|---|---|---|
| $u_1$ | 1 | 1 | 0 |
| $u$ | 0 | 0 | 0 |

binary preds.

| $n$ | $u_1$ | $u$ |
|---|---|---|
| $u_1$ | 0 | 1/2 |
| $u$ | 0 | 1/2 |

| $eq$ | $u_1$ | $u$ |
|---|---|---|
| $u_1$ | 1 | 0 |
| $u$ | 0 | 1/2 |

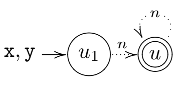


**Fig. 3.** Example showing how the imprecision that was illustrated in Fig. 1 is avoided with the predicate-maintenance formula $\mu_{is[n],\mathtt{y}=\mathtt{x}}(v) = is[n](v)$. (Ex. 2 shows how this is generated automatically.)

To sum up, in past incarnations of our work, the user must supply a formula $\mu_{p,st}$ for each instrumentation predicate $p$ and each statement $st$. In effect, the user must write down two *separate* characterizations of each instrumentation predicate $p$: (i) $\psi_p$, 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 is the user's responsibility to ensure that the two characterizations are mutually consistent. In contrast, with the new method for automatically creating predicate-maintenance formulas presented in Sects. 4 and 5, the user's only responsibility is to define the $\psi_p$.

## 4 A Finite-Differencing Scheme for 3-Valued Logic

This section presents a finite-differencing scheme for creating predicate-maintenance formulas. A predicate-maintenance formula $\mu_{p,st}$ for $p \in \mathcal{I}$ is defined in terms of two finite-differencing operators, denoted by $\Delta^-_{st}[\cdot]$ and $\Delta^+_{st}[\cdot]$, which capture the negative and positive changes, respectively, that execution of statement $st$ induces in an instrumentation predicate's value. The formula $\mu_{p,st}$ is created by combining $p$ with $\Delta^-_{st}[\psi_p]$ and $\Delta^+_{st}[\psi_p]$ as follows: $\mu_{p,st} = p \;?\; \neg\Delta^-_{st}[\psi_p] : \Delta^+_{st}[\psi_p]$.

Fig. 4 depicts how the static-analysis engine evaluates $\Delta^-_{st}[\psi_p]$ and $\Delta^+_{st}[\psi_p]$ in $S$ 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 Fig. 5. The definitions in Fig. 5 make use of the following operator:

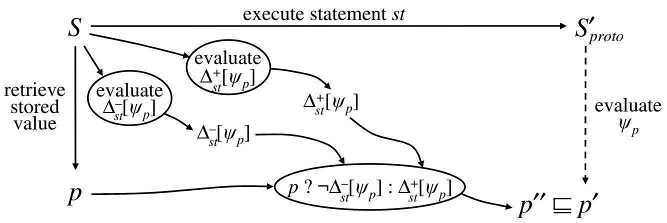


**Fig. 4.** How to maintain the value of $\psi_p$ in 3-valued logic in response to changes in the values of core predicates caused by the execution of statement $st$.

$$\mathbf{F}_{st}[\varphi] \overset{\text{def}}{=} \varphi \;?\; \neg\Delta^-_{st}[\varphi] : \Delta^+_{st}[\varphi]. \tag{6}$$

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

Eqn. (6) and Fig. 5 define a syntax-directed translation scheme that can be implemented via a recursive walk over a formula $\varphi$. The operators $\Delta^-_{st}[\cdot]$ 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 predicates—which are being changed by the $\tau_{c,st}$ formulas; the dependent variable is the value of $\varphi$. A formal justification of Fig. 5 is stated later (Thm. 2); here we merely explain informally a few of the cases from Fig. 5:

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

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

**Fig. 5.** Finite-difference formulas for first-order formulas.

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

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

$\varDelta^+_{st}[p(w_1, \ldots, w_k)] = (\exists\, v\colon \varDelta^+_{st}[\varphi_1]) \wedge \neg p$, if $p \in \mathcal{I}$ and $\psi_p \equiv \exists\, v\colon \varphi_1$. This is similar to the previous case, 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 $(\exists\, v\colon \varDelta^+_{st}[\varphi_1]) \wedge \neg p$ is evaluated, the stored value of $\exists\, v\colon \varphi_1$, i.e., $p$, will be used instead of the value obtained by reevaluating $\exists\, v\colon \varphi_1$.

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

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

*Example 2.* Consider the instrumentation predicate $is[n]$ ("is-shared using n fields"), defined in Eqn. (5). Fig. 6 shows the formulas obtained for $\varDelta^+_{st}[is[n](v)]$ and $\varDelta^-_{st}[is[n](v)]$.

For a particular statement, the formulas in Fig. 6 can usually be simplified. For instance, for y = x, the predicate-transfer formula $\tau_{n,\mathtt{y=x}}(v_1, v_2)$ is $n(v_1, v_2)$; see Fig. 1. Thus, by Fig. 5, the formulas for $\varDelta^-_{\mathtt{y=x}}[n(v_1, v)]$ and $\varDelta^+_{\mathtt{y=x}}[n(v_1, v)]$ are both $n(v_1, v) \wedge \neg n(v_1, v)$, which simplifies to $\mathbf{0}$. (In our implementation, simplifications are performed greedily at formula-construction time; e.g., the constructor for $\wedge$ rewrites $\mathbf{0} \wedge p$ to $\mathbf{0}$, $\mathbf{1} \wedge p$ to $p$, $p \wedge \neg p$ to $\mathbf{0}$, etc.) The formulas in Fig. 6 simplify to $\varDelta^+_{\mathtt{y=x}}[is[n](v)] = \mathbf{0}$ and $\varDelta^-_{\mathtt{y=x}}[is[n](v)] = \mathbf{0}$. Consequently, $\mu_{is[n],\mathtt{y=x}}(v) = \mathbf{F}_{\mathtt{y=x}}[is[n](v)] = is[n](v)\ ?\ \neg\mathbf{0} : \mathbf{0} = is[n](v)$. As shown in Fig. 3, this definition of $\mu_{is[n],\mathtt{y=x}}(v)$ avoids the imprecision that was illustrated in Ex. 1.

$$\Delta^+_{st}[is[n](v)] = \left(\exists\, v_1, v_2\colon \left(\begin{array}{c}(\Delta^+_{st}[n(v_1,v)] \wedge \mathbf{F}_{st}[n(v_2,v)])\\ \vee\ (\mathbf{F}_{st}[n(v_1,v)] \wedge \Delta^+_{st}[n(v_2,v)])\end{array}\right) \wedge v_1 \neq v_2\right) \wedge \neg is[n](v)$$

$$\Delta^-_{st}[is[n](v)] = \left\{\begin{array}{l}\left(\exists\, v_1, v_2\colon \left(\begin{array}{c}(\Delta^-_{st}[n(v_1,v)] \wedge n(v_2,v))\\ \vee\ (n(v_1,v) \wedge \Delta^-_{st}[n(v_2,v)])\end{array}\right) \wedge v_1 \neq v_2\right)\\ \wedge\\ \neg\left(\exists\, v_1, v_2\colon \left(\begin{array}{l}(n(v_1,v) \wedge n(v_2,v) \wedge v_1 \neq v_2)\\ ?\ \neg\left(\left(\begin{array}{c}(\Delta^-_{st}[n(v_1,v)] \wedge n(v_2,v))\\ \vee\ (n(v_1,v) \wedge \Delta^-_{st}[n(v_2,v)])\end{array}\right) \wedge v_1 \neq v_2\right)\\ :\ \left(\begin{array}{c}(\Delta^+_{st}[n(v_1,v)] \wedge \mathbf{F}_{st}[n(v_2,v)])\\ \vee\ (\mathbf{F}_{st}[n(v_1,v)] \wedge \Delta^+_{st}[n(v_2,v)])\end{array}\right) \wedge v_1 \neq v_2\end{array}\right)\right)\end{array}\right.$$

**Fig. 6.** Finite-difference formulas for the instrumentation predicate $is[n](v)$.

For 2-STRUCTs, the correctness of the finite-differencing transformation given in Fig. 5 is ensured by the following theorem.

**Theorem 2.** *Let $S$ be a structure in 2-STRUCT, and let $S'_{proto}$ be the proto-structure for statement $st$ obtained from $S$. Let $S'$ be the structure obtained by using $S'_{proto}$ as the first approximation to $S'$ and then filling in instrumentation predicates in a topological ordering of the dependences among them: for each arity-$k$ predicate $p \in \mathcal{I}$, $\iota^{S'}(p)$ is obtained by evaluating $[\![\psi_p(v_1, \ldots, v_k)]\!]_2^{S'}([v_1 \mapsto u'_1, \ldots, v_k \mapsto u'_k])$ for all tuples $(u'_1, \ldots, u'_k) \in (U^{S'})^k$. Then for every formula $\varphi(v_1, \ldots, v_k)$ and complete assignment $Z$ for $\varphi(v_1, \ldots, v_k)$, $[\![\boldsymbol{F}_{st}[\varphi(v_1, \ldots, v_k)]]\!]_2^S(Z) = [\![\varphi(v_1, \ldots, v_k)]\!]_2^{S'}(Z)$.*

For 3-STRUCTs, the soundness of the finite-differencing transformation given in Fig. 5 follows from Thm. 2 by the Embedding Theorem (Thm. 1).

**Malloc and Free.** In [18], the modeling of storage-allocation/deallocation operations is carried out with a two-stage statement 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 predicate tuples that involve the newly allocated individual. Such predicate values are needed for the second stage, in which predicate-transfer formulas for core predicates and predicate-maintenance formulas for instrumentation predicates are applied in the usual fashion, using Eqns. (1) and (3).

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:

$$\begin{tikzcd} \texttt{freelist} \arrow[r] & u_1 \arrow[r, dotted, "n"] & u \arrow[loop above, dotted, "n"] \end{tikzcd} \tag{7}$$

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.

It is true that the use of structure (7) 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 [18] (namely, the *focus* and *coerce* operations), values for predicate tuples that involve the newly materialized individual will always have safe, mutually consistent values.

## 5 Reachability and Transitive Closure

Several instrumentation predicates that depend on **RTC** are shown in Tab. 2.

**Table 2.** Defining formulas of some instrumentation predicates that depend on **RTC**. (Recall that $n^*(v_1, v_2)$ is a shorthand for $(\mathbf{RTC}\ v_1', v_2' \colon n(v_1', v_2'))(v_1, v_2)$.)

| $p$ | **IntendedMeaning** | $\psi_p$ |
|---|---|---|
| $t[n](v_1, v_2)$ | Is $v_2$ reachable from $v_1$ along n fields? | $n^*(v_1, v_2)$ |
| $r[z, n](v)$ | Is $v$ reachable from pointer variable z along n fields? | $\exists\, v_1 \colon z(v_1) \wedge t[n](v_1, v)$ |
| $c[n](v)$ | Is $v$ on a directed cycle of n fields? | $\exists\, v_1 \colon n(v_1, v) \wedge t[n](v, v_1)$ |

Unfortunately, finding a good way to maintain instrumentation predicates defined using **RTC** is challenging because 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 predicates 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 paper, we confine ourselves to an important special case, namely, techniques to maintain instrumentation predicates specified via the **RTC** of a binary formula that defines an acyclic graph. (Some special cases for **RTC** of binary formulas that define possibly-cyclic graphs will be the subject of a future paper.)

Consider a binary instrumentation predicate $p$, 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 $\varphi_1$ is acyclic, it is possible to give a first-order formula that maintains $p$ after the addition or deletion of a single $\varphi_1$-edge. The method we use is a minor modification of a method for maintaining non-reflexive transitive closure in an acyclic graph, due to Dong and Su [7].

In the case of an insertion of a single $\varphi_1$-edge, the maintenance formula is

$$\mathbf{F}_{st}[p](v_1, v_2) = p(v_1, v_2) \vee (\exists\, v_1', v_2' \colon p(v_1, v_1') \wedge \Delta^{+}_{st}[\varphi_1](v_1', v_2') \wedge p(v_2', v_2)). \quad (8)$$

The new value of $p$ contains the old tuples of $p$, as well as those that represent two old paths connected with the new $\varphi_1$-edge.

The maintenance formula to handle the deletion of a single $\varphi_1$-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*):

$$T[p,\varphi_1](v_1,v_2) = (p(v_1,v_2) \wedge \neg S[p,\varphi_1](v_1,v_2)) \vee \mathbf{F}_{st}[\varphi_1](v_1,v_2). \quad (9)$$

Finally, the maintenance formula for $p$ for a single $\varphi_1$-edge deletion is

$$\mathbf{F}_{st}[p](v_1,v_2) = \exists\, v_1', v_2' \colon T[p,\varphi_1](v_1,v_1') \wedge T[p,\varphi_1](v_1',v_2') \wedge T[p,\varphi_1](v_2',v_2). \quad (10)$$

Maintenance formulas (8) and (10) maintain $p$ when two conditions hold: the graph defined by $\varphi_1$ 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 $\varphi_1$-edge deletion is correct, suppose that there is a suspicious tuple $p(u_1,u_k)$, i.e., $S[p,\varphi_1](u_1,u_k) = 1$, but there is a $\varphi_1$-path $u_1,\ldots,u_k$ that does not use the deleted $\varphi_1$-edge. We need to show that $\mathbf{F}_{st}[p](u_1,u_k)$ has the value 1. Suppose that $(a,b)$ is the $\varphi_1$-edge being deleted; because the graph defined by $\varphi_1$ 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 Fig. 7). 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 $\varphi_1$, 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 Eqn. (10).

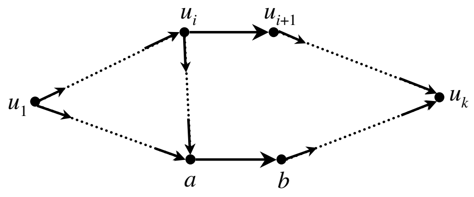


**Fig. 7.** Edge $(a,b)$ is being deleted; $u_i$ is the last node along path $u_1,\ldots,u_i,u_{i+1},\ldots,u_k$ from which $a$ is reachable.

Fig. 8 extends the method for generating predicate-maintenance formulas to handle instrumentation predicates specified via the **RTC** of a binary formula that defines an acyclic graph. Fig. 8 makes use of the operator $T[p,\varphi_1](v,v')$ (Eqn. (9)), but recasts Eqns. (8) and (10) as finite-difference expressions $\Delta^+_{st}[\psi_p]$ and $\Delta^-_{st}[\psi_p]$, respectively.

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 $\varphi_1$, always results in a single edge addition or deletion. If in any admissible 2-STRUCT$[P]$ 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 2-STRUCT$[P]$ 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 these questions 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 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 predicate $q$ has the

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

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

**Fig. 8.** Extension of the finite-differencing method from Fig. 5 to cover **RTC** formulas, for unit-sized changes to an acyclic graph defined by $\varphi_1$.

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 $\varphi_1$. 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 $\varphi_1$ can be updated using the method discussed above.

## 6 Experimental Evaluation

To evaluate the techniques presented in the paper, we extended TVLA to generate predicate-maintenance formulas, and applied it to a test suite of 5 existing analysis specifications, involving 26 programs (see Fig. 9).

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 [10]. The system was used to verify some partial-correctness 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.

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; Non-tree creates a node whose left-child and right-child pointers point to the same subtree.

| Category | Test Program | # of non-id maintenance-formula | | | | Performance | | |
|---|---|---|---|---|---|---|---|---|
| | | schemas | | | instances | Analysis Time (sec.) | | |
| | | total | TC | non-TC | | reference | FD | % increase |
| SLL Shape Analysis | Search | 2 | 0 | 2 | 2 | 0.93 | 0.92 | -0.11 |
| | NullDeref | 2 | 0 | 2 | 3 | 0.96 | 0.96 | 0.31 |
| | GetLast | 3 | 0 | 3 | 4 | 1.14 | 1.14 | -0.44 |
| | DeleteAll | 11 | 2 | 9 | 15 | 0.79 | 0.81 | 3.30 |
| | Reverse | 12 | 2 | 10 | 16 | 1.33 | 1.39 | 4.49 |
| | Create | 11 | 2 | 9 | 21 | 0.73 | 0.76 | 3.68 |
| | Swap | 11 | 2 | 9 | 27 | 0.77 | 0.81 | 5.47 |
| | Delete | 12 | 2 | 10 | 39 | 3.78 | 4.81 | 27.15 |
| | Merge | 11 | 2 | 9 | 64 | 7.69 | 9.34 | 21.50 |
| | Insert | 12 | 2 | 10 | 72 | 3.67 | 4.47 | 21.69 |
| DLL Shape Analysis | Append | 15 | 2 | 13 | 50 | 7.66 | 8.81 | 14.96 |
| | Delete | 16 | 2 | 14 | 74 | 27.97 | 26.87 | -3.93 |
| | Splice | 15 | 2 | 13 | 96 | 3.25 | 3.78 | 16.20 |
| Binary Tree Shape Analysis | Non-tree | 8 | 2 | 6 | 9 | 0.82 | 0.90 | 9.63 |
| | InsertSorted | 13 | 2 | 11 | 43 | 10.08 | 11.19 | 10.98 |
| | Deutsch-Schorr-Waite | 10 | 2 | 8 | 52 | 357.88 | 419.07 | 17.10 |
| | DeleteSorted | 13 | 2 | 11 | 554 | 284.50 | 406.30 | 42.81 |
| SLL Sorting | ReverseSorted | 18 | 2 | 16 | 23 | 1.62 | 1.69 | 4.13 |
| | Bubble | 18 | 2 | 16 | 80 | 36.08 | 41.88 | 16.07 |
| | BubbleBug | 18 | 2 | 16 | 80 | 34.68 | 39.85 | 14.90 |
| | InsertSortBug2 | 18 | 2 | 16 | 87 | 29.95 | 43.52 | 45.29 |
| | InsertSort | 18 | 2 | 16 | 88 | 38.20 | 51.38 | 34.51 |
| | InsertSortBug1 | 18 | 2 | 16 | 88 | 109.91 | 134.15 | 22.05 |
| | MergeSorted | 18 | 2 | 16 | 91 | 12.09 | 14.24 | 17.79 |
| Information Flow | Good Flow | 12 | 2 | 10 | 66 | 58.49 | 67.65 | 15.66 |
| | Bad Flow | 12 | 2 | 10 | 86 | 375.83 | 461.77 | 22.87 |

**Fig. 9.** Results from using hand-crafted vs. automatically generated maintenance formulas for instrumentation predicates.

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 predicate, a *maintenance-formula schema* must be provided. The number of non-identity maintenance-formula schemas is reported in columns 3–5 of Fig. 9, broken down in columns 4–5 into those whose defining formula contains an occurrence of **RTC**, and those that do not. Predicate-maintenance formulas produced by finite differencing are generally larger than the hand-crafted ones. Because this affects analysis time, the number of instances of non-identity maintenance-formula schemas is a meaningful size measure for our experiments. These numbers appear in column 6.

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 and compared the result against the reference answer.

For all 26 test programs, the analysis using automatically generated formulas yielded answers identical to the reference answers.

Columns 7–9 show performance data, which were collected on a 1Ghz AMD Athlon$^{\text{TM}}$ workstation running Red Hat Linux version 7.1. In each case, five runs were made; the longest and shortest times were discarded from each set, and the remaining three averaged. (Figures do not report time spent on loading and initialization, which is not affected by our technique. We also exclude the overhead of formula differencing, because this is not an analysis-time cost.) The geometric mean of the slowdowns when using the automatically generated formulas was approximately 14%, with a median of 15%, mainly due to the fact that the automatically generated formulas are larger than the hand-crafted ones. The maximum slowdown was 45%.[3] 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 predicate-maintenance formulas from Sect. 4 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 predicate-maintenance formula*. For each identity predicate-maintenance formula in the hand-crafted specification, we checked that (after simplification) the corresponding generated predicate-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 non-identity hand-crafted maintenance formula. (The measurements reported in Fig. 9 are based on corrected hand-crafted specifications.)

## 7 Related Work

A weakness of past incarnations of TVLA has been the need for the user to define predicate-maintenance formulas that specify how each statement affects each instrumentation predicate. Recent criticisms of TVLA based on this deficiency are no longer valid [3,15], at least for analyses that can be defined using formulas that define acyclic relations (and also for some classes of formulas that define cyclic relations, using techniques not discussed in this paper). With the algorithm presented in Sects. 4 and 5, the user's responsibility is merely to write the $\psi_p$ formulas; appropriate predicate-maintenance formulas are created automatically.

Graf and Saïdi [8] showed that theorem provers can be used to generate best abstract transformers [4] 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 [3] and other systems [5].) In contrast, the abstract transformers created using the algorithm described in Sects. 4 and 5 are not best transformers; however, this algorithm uses only very simple, linear-time, recursive tree-traversal procedures,

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

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 predicates have been identified.

Paige studied how finite-differencing transformations of applicative set-former expressions could be exploited to optimize loops in very-high-level languages, such as SETL [16]. 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 [13,12]. 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 Sects. 4 and 5 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 Sects. 4 and 5 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 [2] and Sharir [19]. Previous work on incrementally maintaining materialized views in databases [9], "first-order incremental evaluation schemes (FOIES)" [6], and "dynamic descriptive complexity" [17] has also addressed the problem of maintaining one or more auxiliary predicates after new tuples are inserted into or deleted from the base predicates. 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.

Not all finite-differencing transformations that are correct in 2-valued logic (i.e., satisfy Thm. 2), are appropriate for use in 3-valued logic. For instance, Fig. 10 presents an alternative finite-differencing scheme for first-order formulas. In this scheme, $\Delta_{st}[\varphi]$ captures both the negative and positive changes to $\varphi$'s value. With Fig. 10, the maintenance formula for instrumentation predicate $p$ is

$$\mu_{p,st} \stackrel{\text{def}}{=} p \oplus \Delta_{st}[\psi_p], \tag{11}$$

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, Eqn. (11) 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 $S'$, in all successors of $S'$, and so on. With Eqn. (11), $p(u)$ can *never* reacquire a definite value.

In contrast, the maintenance formulas created using the finite-differencing scheme of Fig. 5 do not have this trouble because they have the form $p \,?\, \neg\Delta^-_{st}[\psi_p] : \Delta^+_{st}[\psi_p]$. The use of if-then-else allows $p(u)$ to reacquire a definite value after it has been set to $1/2$: if $p(u)$ 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.

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

**Fig. 10.** An alternative finite-differencing scheme for first-order formulas.

**Acknowledgments.** We thank 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

1. TVLA system. Available at "http://www.math.tau.ac.il/∼rumster/TVLA/".
2. S.B. Akers, Jr. On a theory of Boolean functions. *J. Soc. Indust. Appl. Math.*, 7(4):487–498, December 1959.
3. T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *Conf. on Prog. Lang. Design and Impl.*, New York, NY, 2001. ACM Press.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.
5. S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *Proc. Computer-Aided Verif.*, pages 160–171. Springer-Verlag, July 1999.
6. G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Inf. and Comp.*, 120:101–106, 1995.
7. G. Dong and J. Su. Incremental maintenance of recursive views using relational calculus/SQL. *SIGMOD Record*, 29(1):44–51, 2000.
8. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. Computer-Aided Verif.*, pages 72–83, June 1997.
9. A. Gupta and I.S. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications*. The M.I.T. Press, Cambridge, MA, 1999.
10. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. on Software Testing and Analysis*, pages 26–38, 2000.
11. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.
12. Y.A. Liu, S.D. Stoller, and T. Teitelbaum. Discovering auxiliary information for incremental computation. In *Symp. on Princ. of Prog. Lang.*, pages 157–170, January 1996.
13. Y.A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. of Comp. Program.*, 24:1–39, 1995.
14. K.L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, pages 219–234, 1999.
15. A. Møller and M.I. Schwartzbach. The pointer assertion logic engine. In *Conf. on Prog. Lang. Design and Impl.*, pages 221–231, 2001.
16. R. Paige and S. Koenig. Finite differencing of computable expressions. *Trans. on Prog. Lang. and Syst.*, 4(3):402–454, July 1982.
17. S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *J. Comput. Syst. Sci.*, 55(2):199–209, October 1997.
18. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 24(3):217–298, 2002.
19. M. Sharir. Some observations concerning formal differentiation of set theoretic expressions. *Trans. on Prog. Lang. and Syst.*, 4(2):196–225, April 1982.