Abstraction Refinement for 3-Valued-Logic Analysis

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

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

Abstract. This paper concerns the question of how to create abstractions that are useful for program analysis. It presents a method that refines an abstraction automatically for analysis problems in which the semantics of statements and the query of interest are expressed using logical formulas. Refinement is carried out by introducing new instrumentation relations (defined via logical formulas over core relations, which capture the basic properties of memory configurations). A tool that incorporates the algorithm has been implemented and applied to several algorithms that manipulate linked lists and binary-search trees. In all but a few cases, the tool is able to demonstrate (i) the partial correctness of the algorithms, and (ii) that the algorithms possess additional properties—e.g., stability or antistability.

1 Introduction

This paper presents a technique for automatically creating abstractions for use in program analysis. As in some previous work [12, 19, 5, 13, 16, 8, 9, 3, 4, 11], the technique involves the successive refinement of the abstraction in use. However, unlike previous work, the work presented in this paper is aimed specifically at programs that manipulate pointers and heap-allocated data structures.

The paper presents an abstraction-refinement method for use in static analyses based on 3-valued logic [20], where the semantics of statements and the query of interest are expressed using logical formulas. Refinement is performed by introducing new instrumentation relations (defined via logical formulas over core relations, which capture the basic properties of memory configurations). The algorithm presented here analyzes the sources of imprecision in the evaluation of the query, and chooses how to define new instrumentation relations using subformulas of the query.

In this 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. 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 query is always satisfied 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 [15, 2].

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 consistency of the 2-valued and 3-valued viewpoints is ensured by a basic theorem that relates the two logics [20]. 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. They provide a mechanism to fine-tune an abstraction: an instrumentation relation, which is defined by a logical formula over the core relation symbols, captures a property that an individual memory cell may or may not possess. In general, adding additional instrumentation relations 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.

The choice of formulas to be used as definitions of instrumentation relations is crucial to the precision, as well as the cost, of the analysis. Until now, TVLA users have been faced with the task of identifying an instrumentation-relation set that gives them a definite answer to the query, but does not make the cost prohibitive. This was arguably the key remaining challenge in the TVLA user-model.

The contributions of this work can be summarized as follows:

- It is a step towards automatically generating useful abstractions for static analyses based on 3-valued logic.
- Essentially all of the user-level obligations for which TVLA has been criticized in the past have been eliminated. The input required to specify a program analysis consists of
  - a program (at present, in the form of a transition system)
  - a query (i.e., a formula that characterizes the acceptable outputs)
  - a characterization of the program's allowable inputs.
- The technique has been implemented as an extension of TVLA.
- We present experimental evidence that the use of this approach in an iterative abstraction-refinement loop can yield precise answers to queries. We illustrate the technique by testing sortedness, stability, and antistability queries on a collection of programs that perform destructive list manipulation, as well as by testing partial correctness of two binary-search-tree algorithms.

As a methodology for verifying properties of programs, the advantages of the approach taken in the paper are:

- No loop invariants are required.
- No theorem provers are involved, and thus every refinement step must terminate.
- The method is based on abstract interpretation, and thus the entire process must terminate.
- The method applies to static analyses based on 3-valued first-order logic, and hence it applies to programs that manipulate pointers and heap-allocated data structures, and eliminates the need for the user to write the usual proofs required for abstract interpretation—i.e., to demonstrate that the abstract structures that the analyzer manipulates correctly model the concrete heap-allocated data structures that the program manipulates.

The remainder of the paper is organized as follows: §2 introduces terminology and notation. Readers familiar with TVLA can skip to §3, which illustrates our goals on the problem of verifying the partial correctness of a sorting routine. §4 describes a method

for iterative abstraction refinement and illustrates the method on the example of §3. §5 presents experimental results. §6 discusses related work.

2 Modeling and Abstracting the Heap with Logical Structures

```
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?
$q(v)$	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?
$dle(v_1, v_2)$	Is the `data` field of v_1 less than or equal to that of v_2?

(b)

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

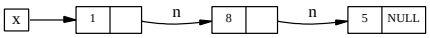
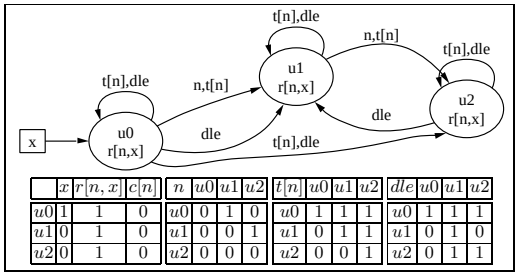


Fig. 1. A possible store for a linked list.

This section summarizes the shape-analysis framework described in [20]. In this approach, concrete memory configurations or *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, Tab. 1 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 Fig. 1. 2-valued logical structures then represent memory configurations: the individuals 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. 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 Tab. 1; 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 [10]. Fig. 2 shows 2-valued structure S_2, which represents the store of Fig. 1 (relations $t[n]$, $r[n, x]$, and $c[n]$ are explained in §2.2).



	x	$r[n,x]$	$c[n]$
$u0$	1	1	0
$u1$	0	1	0
$u2$	0	1	0

n	$u0$	$u1$	$u2$
$u0$	0	1	0
$u1$	0	0	1
$u2$	0	0	0

$t[n]$	$u0$	$u1$	$u2$
$u0$	1	1	1
$u1$	0	1	1
$u2$	0	0	1

dle	$u0$	$u1$	$u2$
$u0$	1	1	1
$u1$	0	1	0
$u2$	0	1	1

Fig. 2. A logical structure S_2 that represents the store shown in Fig. 1 in graphical and tabular representations.

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 relation like $x(v)$ of Tab. 1 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.

Let $\mathcal{R} = \{eq, p_1, \ldots, p_n\}$ be a finite vocabulary of relation symbols, where $\mathcal{R}_k$ denotes the set of relation symbols of arity k (and $eq \in \mathcal{R}_2$). The set of 2-valued structures is denoted by $\mathcal{S}_2[\mathcal{R}]$. The syn-

tax of first-order formulas with equality and reflexive transitive closure is defined as follows:

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

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

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 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: $(v_1 = v_2) \stackrel{\text{def}}{=} eq(v_1, v_2)$; $(v_1 \neq v_2) \stackrel{\text{def}}{=} \neg eq(v_1, v_2)$; $(\varphi_1 \rightarrow \varphi_2) \stackrel{\text{def}}{=} (\neg\varphi_1 \vee \varphi_2)$; $(\varphi_1 \leftrightarrow \varphi_2) \stackrel{\text{def}}{=} (\neg\varphi_1 \vee \varphi_2) \wedge (\neg\varphi_2 \vee \varphi_1)$; and for a binary relation p, $p^*(v_1, v_2) \stackrel{\text{def}}{=} (\mathbf{RTC}\ v_1', v_2'\colon p(v_1', v_2'))(v_1, v_2)$. The order of precedence among the connectives, from highest to lowest, is as follows: $\neg$, $\wedge$, $\vee$, $\forall$, and $\exists$.

Definition 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 of arity k to a truth-valued function: $\iota^S(p)\colon (U^S)^k \rightarrow \{0, 1\}$. 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 $Z\colon \{v_1, v_2, \ldots\} \rightarrow U^S$).

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 shown in Fig. 3.

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

$$
[\![(\mathbf{RTC}\ v_1', v_2'\colon \varphi_1)(v_1, v_2)]\!]_2^S(Z)
$$

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

Fig. 3. The (2-valued) meaning of a formula φ.

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

2.1 Program Analysis Via 3-Valued Logic

In 3-valued logic, formulas 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 $l_2 = 1/2$. The symbol $\sqcup$ denotes the least-upper-bound operation with respect to $\sqsubseteq$.

Definition 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 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 $u \in U^S$, $\iota^S(\mathit{eq})(u, u) \sqsupseteq 1$, and (ii) for all $u_1, u_2 \in U^S$ such that u_1 and u_2 are distinct individuals, $\iota^S(\mathit{eq})(u_1, u_2) = 0$.

For an assignment Z, the *(3-valued) meaning* of a formula φ, denoted by $[\![\varphi]\!]^S_3(Z)$, yields a truth value in $\{0, 1/2, 1\}$. The meaning of φ is defined exactly as in Defn. 2, but interpreted over $\{0, 1/2, 1\}$. S and Z *potentially satisfy* φ if $[\![\varphi]\!]^S_3(Z) \sqsupseteq 1$. The set of 3-valued structures is denoted by $\mathcal{S}_3[\mathcal{R}]$.

An individual for which $\iota^S(\mathit{eq})(u, u) = 1/2$ is called a *summary individual*. In the program-analysis context, a summary individual abstracts one or more fragments of a data structure, and can represent more than one concrete memory cell.

A concrete operational semantics is defined by specifying a structure transformer for each kind of edge e that can appear in a CFG. A structure transformer is specified by providing a collection of *relation-update formulas*, $c(v_1, \ldots, v_k) = \tau_{c,e}(v_1, \ldots, v_k)$, one for each core relation c. These formulas define how the core relations of a logical structure S that arises at the source of e are transformed by e to create a logical structure S' at the target of e; typically, they define the value of relation c in S' as a function of c's value in S. Edge e may optionally have a *precondition formula*, which filters out structures that should not follow the transition along e. The postcondition operator $post$ for edge e is defined by lifting e's structure transformer to sets of structures.

The collecting semantics of a program corresponds to a postcondition operator of type $\wp(\mathcal{S}_2) \to \wp(\mathcal{S}_2)$. However, $\wp(\mathcal{S}_2)$ is not suitable as an abstract domain; for instance, when the programming language being modeled supports allocation from the heap, the set of individuals that may appear in a structure is unbounded (or so large that it is the same from a practical point of view), and thus there is no a priori upper bound on the cardinality of elements of $\wp(\mathcal{S}_2)$.

One can sidestep this problem by abstracting sets of 2-valued structures using 3-valued structures equipped with a suitable order [20]. A set of stores is then represented by a (finite) set of 3-valued logical structures.

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 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 meaning of a formula is preserved by embedding in the following sense: 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 1. (Embedding Theorem [20, 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 assignment Z that is defined on all free variables of φ, $[\![\varphi]\!]^S_3(Z) \sqsubseteq [\![\varphi]\!]^{S'}_3(f \circ Z)$.*

To obtain a computable abstract domain, we need a way to ensure that the 3-valued structures used to represent memory configurations are always of finite size. We do this

by defining an equivalence relation between individuals, and considering the (finite) 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:

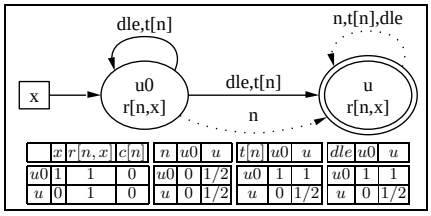
Definition 6 (Canonical Abstraction). Let $S = (U, \iota) \in \mathcal{S}_2$, and let $\mathcal{A} \subseteq \mathcal{P}_1$ be some chosen subset of the unary relation symbols. The relations in $\mathcal{A}$ are called *abstraction relations*; they define the following equivalence relation $\simeq_{\mathcal{A}}$ on U:

$$u_1 \simeq_{\mathcal{A}} u_2 \iff \text{forall } p \in \mathcal{A}, \iota(p)(u_1) = \iota(p)(u_2),$$

and the surjective function $f_{\mathcal{A}} : U \to U/\simeq_{\mathcal{A}}$, 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 $\mathcal{A}$ is the structure $f_{\mathcal{A}}(S)$.

If all unary relations are abstraction relations ($\mathcal{A} = \mathcal{R}_1$), the canonical abstraction of 2-valued logical structure S_2 is S_4, shown in Fig. 4, with $f_{\mathcal{A}}(u_0) = u_0$ and $f_{\mathcal{A}}(u_1) = f_{\mathcal{A}}(u_2) = u$. S_4 represents all lists with two or more elements, in which the first element's `data` value is lower than the `data` values in the rest of the list.



	x	$r[n,x]$	$c[n]$
$u0$	1	1	0
u	0	1	0

n	$u0$	u
$u0$	0	1/2
u	0	1/2

$t[n]$	$u0$	u
$u0$	1	1
u	0	1/2

dle	$u0$	u
$u0$	1	1
u	0	1/2

Fig. 4. A 3-valued structure S_4 that is the canonical abstraction of structure S_2.

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

- Individuals are represented by circles containing their names and values for unary relations (0 values are usually omitted).
- A summary individual is represented by a double circle.
- A unary relation p corresponding to a pointer program variable is represented by a solid arrow from p to the 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 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 $\iota(q)(u_i, u_j) = 1$, and by the absence of a q-arrow between pairs u'_i and u'_j for which $\iota(q)(u'_i, u'_j) = 0$.
- Binary relations with value $1/2$ are represented by dotted arrows.

Canonical abstraction ensures that each 3-valued structure is no larger than some fixed size, known *a priori*. Moreover, a given formula is interpreted consistently in both the concrete domain (namely, $\wp(\mathcal{S}_2)$) and the abstract domain ($\wp(\mathcal{S}_3)$). Thanks to the Embedding Theorem, the meaning of the two interpretations is consistent with respect to the abstraction, although the value of a formula on an abstract structure $\alpha(S)$ may be less precise than its value on the concrete structure S. Consequently, for each kind of statement in the programming language, the structure transformers for the abstract semantics—and hence the abstract postcondition operator—can again be defined via a collection of formulas; in fact, the abstract transformers are defined by the *same* relation-update formulas that define the concrete semantics.

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 collection of equations on variables that take their values in $\wp(\mathcal{S}_3)$ [20].

2.2 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 relation symbol $p \in \mathcal{I} \subseteq \mathcal{R}_k$ is defined by an *instrumentation-relation definition formula* $\psi_p(v_1, \ldots, v_k)$. Instrumentation relation symbols may appear in the defining formulas of other instrumentation relations as long as there are no circular dependences.

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

Table 2. Defining formulas of some commonly used instrumentation relations. There is a separate relation $r[n, x]$ for every program variable x. (Recall that $n^*(v_1, v_2)$ is a shorthand for (**RTC** $v_1', v_2' \colon n(v_1', v_2'))(v_1, v_2)$.)

The introduction of unary instrumentation relations that are then 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** (transitive closure), 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. Tab. 2 lists some instrumentation relations that are important for the analysis of programs that use type `List`.

We are sometimes interested in making assertions that compare the state of a store at the end of a procedure with its state at the start. For instance, we may be interested in checking that all list elements reachable from variable x at the start of a procedure are guaranteed to be reachable from x at the end. To allow the user to make such assertions, we double the vocabulary: for each relation p, we extend the program-analysis specification with a *history* relation, p_0, which serves as an indelible record of the state of the store at the entry point. We will use the term history relations to refer to the latter kind of relations, and the term *active* relations to refer to the relations from the original vocabulary. We can now express the property mentioned above:

$$\forall v : r_0[n, x](v) \leftrightarrow r[n, x](v). \tag{1}$$

If it evaluates to 1, then the elements reachable from x after the procedure executes are exactly the same as those reachable at the beginning of the procedure, and consequently the procedure performs a permutation of list x.

In addition to history relations, we introduce a collection of nullary instrumentation relations that track whether active relations have changed from their initial values. For each active relation $p(v_1, \ldots, v_k)$, the relation $same[p]()$ is defined by $\psi_{same[p]} = \forall v_1, \ldots, v_k : p(v_1, \ldots, v_k) \leftrightarrow p_0(v_1, \ldots, v_k)$. We can now use $same[r[n, x]]()$ in place of Formula (1) when asserting the permutation property.

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 should obtain their values in some other way; in particular, after a transition from structure S to S' via transformer τ, the new value for an instrumentation relation p should be computed incrementally from the known value of p in S. An algorithm that uses τ and p's defining formula $\psi_p(v_1, \ldots, v_k)$ to generate an appropriate incremental *relation-maintenance formula* $\mu_{p,\tau}$ is given in [18].

3 Example: Specifying and Verifying Sortedness

Given the static-analysis algorithm defined in the preceding section, to demonstrate the partial correctness of a procedure, the user must supply the following program-specific information:

- The procedure's control-flow graph.
- A set of 3-valued structures that characterize the acceptable inputs.
- A query; i.e., a formula that characterizes the intended outputs.

The initial 3-valued structures are supplied to the analysis algorithm as the abstract value for the procedure's entry point; the analysis algorithm is then run; finally, the query is evaluated on the structures that are generated at the exit point.

Consider the problem of establishing that the version of `InsertSort` shown in Fig. 5 is partially correct. Fig. 6 shows the three structures that characterize the set of stores in which program variable `x` points to an acyclic linked list. After running the analysis of `InsertSort`, we would check to see whether, for all of the structures that arise at the procedure's exit node, the following formula evaluates to 1:

```
[1] void InsertSort(List x) {
[2]  List r, pr, rn, l, pl;
[3]  r = x;
[4]  pr = NULL;
[5]  while (r != NULL) {
[6]   l = x;
[7]   rn = r->n;
[8]   pl = NULL;
[9]   while (l != r) {
[10]   if (l->data > r->data) {
[11]    pr->n = rn;
[12]    r->n = l;
[13]    if (pl == NULL) x = r;
[14]    else pl->n = r;
[15]    r = pr;
[16]    break;
[17]   }
[18]   pl = l;
[19]   l = l->n;
[20]  }
[21]  pr = r;
[22]  r = rn;
[23] }
[24]}
```

Fig. 5. Stable version of insertion sort.

$$\forall\, v_1 : r[n, x](v_1) \rightarrow (\forall\, v_2 : n(v_1, v_2) \rightarrow dle(v_1, v_2)). \tag{2}$$

If the formula evaluates to 1, then the nodes reachable from `x` must be in non-decreasing order.

An astute reader will notice that a "sorting" procedure that always returns `NULL` will satisfy Formula (2) at the exit point! Thus, Formula (2) is only part of the specification of the post-condition of a correct sorting procedure. A second property required of a correct sorting procedure (as well as of many other procedures that manipulate sorted linked lists) is that the output list must be a permutation of the input list. This can be established by using Formula (1) (see §2).

Fig. 2 shows 2-valued structure S_2, in which the middle list node has a larger `data`-value than the other two nodes (one of the stores that this structure represents is shown in Fig. 1). Note that our (correct) implementation of insertion sort cannot produce the store of Fig. 1, and so S_2 is not an accurate representation of the stores that can arise at line [24] of Fig. 5. Given the structures shown in Fig. 6 as the abstract input structures, abstract interpretation collects 3-valued structure S_4 shown in Fig. 4 at line [24]. Note

that Formula (2) evaluates to $1/2$ on S_4. While the first list element is guaranteed to be in correct order with respect to the remaining elements — note the definite dle edge between the first node and the summary node — there is no guarantee that all list nodes represented by the summary node are in correct order. In particular, because S_4 represents S_2, it admits the store of Fig. 1 as a possible output. Thus, the abstraction that we used was insufficiently fine-grained to establish partial correctness of `InsertSort`. In fact, the abstraction is insufficiently fine-grained to separate the set of sorted lists from the set of lists not in sorted order.

In [14], Lev-Ami et al. used TVLA to establish the partial correctness of `InsertSort`. The key step was the introduction of instrumentation relation $inOrder[dle, n](v)$, which holds for nodes whose `data`-components are less than or equal to those of their `n`-successor; $inOrder[dle, n](v)$ was defined by:

$$inOrder[dle, n](v) \stackrel{\text{def}}{=} \forall v_1 : n(v, v_1) \rightarrow dle(v, v_1). \tag{3}$$

The sortedness property was then stated as follows (cf. Formula (2)):

$$\forall v : r[n, x](v) \rightarrow inOrder[dle, n](v). \tag{4}$$

After the introduction of relation $inOrder[dle, n]$, the 3-valued structures that are collected by abstract interpretation at the end of `InsertSort` describe all possible stores in which variable `x` points to an acyclic, *sorted* linked list. In all of these 3-valued structures, Formulas (4) and (1) both evaluate to 1 (assuming, in the case of Formula (1), that instrumentation relation $r_0[n, x]$ was added to the analysis). Consequently, `InsertSort` is guaranteed to work correctly on all acceptable inputs.

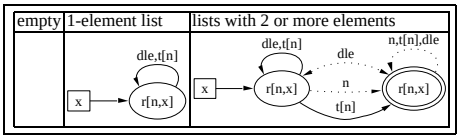


Fig. 6. The structures that describe possible inputs to `InsertSort`.

4 Iterative Abstraction Refinement

In [14], instrumentation relation $inOrder[dle, n]$ was defined explicitly (by the TVLA user). Heretofore, there have really been two burdens placed on the TVLA user:

(i) he must have insight into the behavior of the program, and

(ii) he must translate that insight into logical notation by formulating appropriate instrumentation-relation definition formulas (e.g., Formula (3)).

The goal of the present paper is to automate the introduction of instrumentation relations, such as $inOrder[dle, n]$. In the case of `InsertSort`, the goal is to obtain definite answers when evaluating Formula (2) on the structures collected by abstract interpretation at line [24] of Fig. 5. Fig. 7 gives pseudo-code for our technique, the steps of which can be explained as follows:

- (Line [1]; §4.3) Use a *data-structure constructor* to compute the abstract input structures that represent all valid inputs to the program.
- Perform an abstract interpretation to collect a set of structures at each program point and evaluate the query on the structures at exit. If a definite answer is obtained on all structures, terminate. Otherwise, perform abstraction refinement:
  - (Line [6]; §4.1) Identify subformulas of the query that are responsible for the imprecision, and use them to define new instrumentation relations.

- (Line [7]; §4.2) Replace all occurrences of these subformulas in the query and in the definitions of other instrumentation relations with the use of the corresponding new instrumentation relation symbols, and apply finite differencing to obtain relation-maintenance formulas for the newly introduced instrumentation relations, as well as for those instrumentation relations whose definitions have been changed.
- (Line [8]; §4.3) Obtain the most precise possible values for the newly introduced instrumentation relations in abstract structures that define the valid inputs to the program. This is achieved by "reconstructing" the valid inputs by performing abstract interpretation of the data-structure constructor.

Because a query has finitely many subformulas, the number of abstraction-refinement steps is finite. Because, additionally, each run of the analysis explores a bounded number of 3-valued structures, the algorithm is guaranteed to terminate.

```
Input: the program's transition relation,
       a data-structure constructor,
       a query φ (a closed formula)
[1]  Construct abstract input
[2]  do
[3]    Perform abstract interpretation
[4]    Let S_1,...,S_k be the set of
        3-valued structures at exit
[5]    if for all S_i, [[φ]]_3^{S_i}([]) ≠ 1/2 break
[6]    Find formulas ψ_{p_1},...,ψ_{p_k} for new
        instrumentation relations p_1,...,p_k
[7]    Refine the actions that define
        the program's transition relation
[8]    Refine the abstract input
[9]  while(true)
```

Fig. 7. Pseudo-code for iterative abstraction refinement.

4.1 Instrumentation Relation Generation

A first attempt at abstraction refinement could be the introduction of the query itself as a new instrumentation relation. However, this usually does not lead to a definite answer to the query. For instance, with `InsertSort`, introducing the query as a new instrumentation relation is ineffective because no statement of the program has the effect of changing the value of such an instrumentation relation from $1/2$ to 1.

However, as we saw in §3, the introduction of unary instrumentation relation $inOrder[dle, n]$ allows the sortedness query to be established. When $inOrder[dle, n]$ is present, there are several statements of the program where abstract interpretation results in new definite entries for $inOrder[dle, n]$. For instance, in lines [12]–[14] of Fig. 5, the insertion of the node pointed to by `r` (say u) before the node pointed to by `l`, results in a new definite entry $inOrder[dle, n](u)$.

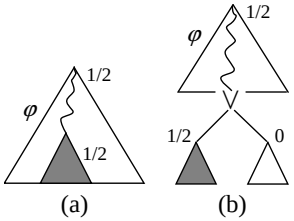


Fig. 8. (a) Recursive-descent function $instrum$ finds the subformulas of φ that can cause the $1/2$ answer. (b) Example: $\vee$ formula.

An algorithm to generate new instrumentation relations should take into account the sources of imprecision. When evaluating the query, the algorithm can identify the subformulas that are responsible for the indefinite answer. These subformulas are good candidates to define new instrumentation relations. Fig. 9 presents function $instrum$, a recursive-descent procedure to generate defining formulas for new instrumentation relations. The arguments to the function are formula φ, logical structure $S \in \mathcal{S}_3[\mathcal{R}]$, and an assignment Z that is defined on all free variables of φ. In the top-level invocation, φ is the (nullary) query, Z is empty, and S is a structure collected at the exit node by the last run of abstract interpretation for which $[\![\varphi]\!]_3^S(Z) = 1/2$.

φ	return value of $instrum(\varphi, S, Z)$	other actions
$\mathbf{0}, \mathbf{1}$	*ERROR*	
$\mathbf{1/2}$	$\emptyset$	
$v_1 = v_2$	$\emptyset$	
$p(v_1, \ldots, v_k)$	$(p \in \mathcal{C})\ ?\ \emptyset : instrum(\psi_p, S, Z)$	$if\ \ (k = 1 \wedge p \notin \mathcal{A})$ $\mathcal{A} := \mathcal{A} \cup \{p\}$
$\neg\varphi_1$	$instrum(\varphi_1, S, Z)$	
$\varphi_1 \vee \varphi_2$ $\varphi_1 \wedge \varphi_2$	$(\varphi \notin \{\psi_p \mid p \in \mathcal{I}\})\ ?\ \{\varphi\} : \emptyset$ $\cup\ ([\![\varphi_1]\!]_3^S(Z) = 1/2)\ ?\ instrum(\varphi_1, S, Z) : \emptyset$ $\cup\ ([\![\varphi_2]\!]_3^S(Z) = 1/2)\ ?\ instrum(\varphi_2, S, Z) : \emptyset$	
$\exists\, v\colon \varphi_1$ $\forall\, v\colon \varphi_1$	$(\varphi \notin \{\psi_p \mid p \in \mathcal{I}\})\ ?\ \{\varphi\} : \emptyset$ $\cup \bigcup_{u \in S} ([\![\varphi_1]\!]_3^S(Z[v \mapsto u]) = 1/2$ $?\ instrum(\varphi_1, S, Z[v \mapsto u])$ $: \emptyset)$	
RTC $v_1', v_2'\colon \varphi_1$	$(\varphi \notin \{\psi_p \mid p \in \mathcal{I}\})\ ?\ \{\varphi\} : \emptyset$ $\cup \bigcup_{u_1', u_2' \in S,\ u_1' \neq u_2'} ([\![\varphi_1]\!]_3^S(Z[v_1' \mapsto u_1', v_2' \mapsto u_2']) = 1/2)$ $?\ instrum(\varphi_1, S, Z[v_1' \mapsto u_1', v_2' \mapsto u_2'])$ $: \emptyset)$	

Fig. 9. Function $instrum$, which looks for formulas to be used as definitions of new instrumentation relations.

A precondition of $instrum$ is that $[\![\varphi]\!]_3^S(Z) = 1/2$. Starting with this assumption, $instrum$ attempts to find subformulas of φ that, if sharpened, would sharpen the value of the whole formula (see Figs. 8 (a) and (b)). If such subformulas are found, they will be used to define new instrumentation relations. Below are explanations of a few cases:

$instrum(\mathbf{1}, \ldots)$ This violates the precondition of $instrum$.

$instrum(\mathbf{1/2}, \ldots)$ Nothing can be done in this case.

$instrum(p \in \mathcal{C}, \ldots)$ If p is unary and is not in the set of abstraction relations, add it to the set of abstraction relations.

$instrum(p \in \mathcal{I}, \ldots)$ Try to sharpen the definition of p, i.e., ψ_p. Also, if p is unary and is not in the set of abstraction relations, add it to the set of abstraction relations.

$instrum(\varphi_1 \vee \varphi_2, \ldots)$ If φ ($\varphi_1 \vee \varphi_2$) does not define an instrumentation relation, it will be used as the definition of a new instrumentation relation. Also, inspect φ_1 and φ_2 to find subformulas that can cause φ to evaluate to $1/2$.

$instrum(\exists\, v : \varphi_1, S, Z)$ If φ ($\exists\, v\ :\ \varphi_1$) does not define an instrumentation relation, it will be used as the definition of a new instrumentation relation. Also, inspect φ_1 under different bindings $v \mapsto u$ to find subformulas of φ_1 that can cause φ to evaluate to $1/2$.

Each formula φ returned by $instrum$ is given a name (say q) and used as the definition of a new instrumentation relation $q(v_1, \ldots, v_k)$, where $v_1, \ldots, v_k$ are the free variables of φ (in order of their appearance in the formula). All new unary instrumentation relations are added as non-abstraction relations. However, they may be added to the set of abstraction relations $\mathcal{A}$ on a subsequent iteration (see column three of entry $p(v_1, \ldots, v_k)$ in Fig. 9, which handles core and instrumentation relations).

p	ψ_p
$sorted_1()$	$\forall\, v_1 : r[n, x](v_1) \rightarrow (\forall\, v_2 : n(v_1, v_2) \rightarrow dle(v_1, v_2))$
$sorted_2(v_1)$	$r[n, x](v_1) \rightarrow (\forall\, v_2 : n(v_1, v_2) \rightarrow dle(v_1, v_2))$
$sorted_3(v_1)$	$\forall\, v_2 : n(v_1, v_2) \rightarrow dle(v_1, v_2)$
$sorted_4(v_1, v_2)$	$n(v_1, v_2) \rightarrow dle(v_1, v_2)$

Table 3. Instrumentation relations created after the call to $instrum$.

Example. As we saw in §3, abstract interpretation collects 3-valued structure S_4 of Fig. 4 at the exit node of `InsertSort`.[3] The sortedness query (Formula (2)) evaluates to $1/2$ on S_4, triggering a call to $instrum$ with Formula (2), structure S_4, and empty assignment Z, as arguments.

Tab. 3 shows the instrumentation relations that were created as a result of the call to $instrum$ on the first iteration of abstraction refinement. Note that $sorted_3$ is defined exactly as $inOrder[dle, n]$, which was the key insight for the results of [14]. Note also that $instrum$ returned no subformulas of the definition of $r[n, x]$. This is because $r[n, x](v)$ evaluates to a definite value (1) for both $v \mapsto u$ and $v \mapsto u_0$ (see Fig. 4).

4.2 Refinement of the Actions that Define the Program's Transition Relation

The actions that define the program's transition relation need to be modified to gain precision improvements from storing and maintaining the new instrumentation relations. To this end, for each new instrumentation relation $p(v_1, \ldots, v_k)$, the query and all other instrumentation relations' defining formulas are scanned for occurrences of ψ_p. Every occurrence of $\psi_p\{w_1/v_1, \ldots, w_k/v_k\}$, i.e., ψ_p with w_i substituted for free variable v_i, is replaced with $p(w_1, \ldots, w_k)$, thus enabling the use of stored value $p(w_1, \ldots, w_k)$ in place of the evaluation of ψ_p.

p	ψ_p
$sorted_1()$	$\forall\, v_1 : sorted_2(v_1)$
$sorted_2(v_1)$	$r[n, x](v_1) \rightarrow sorted_3(v_1)$
$sorted_3(v_1)$	$\forall\, v_2 : sorted_4(v_1, v_2)$
$sorted_4(v_1, v_2)$	$n(v_1, v_2) \rightarrow dle(v_1, v_2)$

Table 4. Final version of instrumentation relations introduced by abstraction refinement.

To complete transition-relation refinement, finite differencing creates relation-maintenance formulas for the new instrumentation relations, as well as for those instrumentation relations whose definitions have been changed. This improves the precision with which relations' stored values are maintained during abstract interpretation [18].

Example. During transition-relation refinement of `InsertSort`, the use of Formula (2) in the query is replaced with the use of the stored value $sorted_1()$. Then the definitions of all instrumentation relations are scanned for occurrences of $\psi_{sorted_1}, \ldots, \psi_{sorted_4}$ (in that order). These occurrences are replaced with names of the four relations. In this case, only the new relations' definitions are changed, yielding the definitions of Tab. 4.

4.3 Refinement of the Abstract Input

Before performing abstract interpretation of the refined program, we need to update the abstract structures that characterize the acceptable inputs to the procedure with values for the new instrumentation relations. To gain maximum benefit from maintaining $p(v_1, \ldots, v_k)$, abstract interpretation needs to start with the most precise possible values for p in abstract input structures. While simply evaluating ψ_p on abstract input structures for all assignments to free variables $v_1, \ldots, v_k$ results in safe values, these values are likely to be imprecise.

We illustrate the issue on the stability property. This property usually arises in the context of sorting procedures, but actually applies to list-manipulating programs in general: the stability query (Formula (5)) asserts that the relative order of elements with equal `data`-components remains the same.

$$\forall\, v_1, v_2 : (dle(v_1, v_2) \wedge dle(v_2, v_1) \wedge t_0[n](v_1, v_2)) \rightarrow t[n](v_1, v_2) \qquad (5)$$

[3] In our implementation, a given round of abstract interpretation is stopped as soon as imprecision is detected.

The first run of abstract interpretation on procedure `InsertSort` does not result in a definite answer to the stability query. The first round of abstraction refinement then introduces the following subformula of Formula (5) as a new instrumentation relation:

$$stable_2(v_1, v_2) \stackrel{\text{def}}{=} (dle(v_1, v_2) \wedge dle(v_2, v_1) \wedge t_0[n](v_1, v_2)) \rightarrow t[n](v_1, v_2) \qquad (6)$$

Consider the rightmost structure of Fig. 6,[4] which includes one concrete and one summary individual; call them u_c and u_s, respectively. If we simply evaluate Formula (6) on the structure, we obtain the definite value 1 for tuples (u_c, u_c), (u_c, u_s), and (u_s, u_c). However, the evaluation yields value $1/2$ for tuple (u_s, u_s) because $dle(u_s, u_s)$, $t_0[n](u_s, u_s)$, and $t[n](u_s, u_s)$ all equal $1/2$.

Our methodology for obtaining values for abstract input structures is to perform an abstract interpretation on a loop that constructs the family of all valid inputs to the program (we call such a loop a **D**ata-**S**tructure **C**onstructor, or DSC). This allows the values of instrumentation relations to be maintained (as input structures are manufactured from the empty store) rather than computed; in general, this results in more precise values for the instrumentation relations. Fig. 10 illustrates the idea.

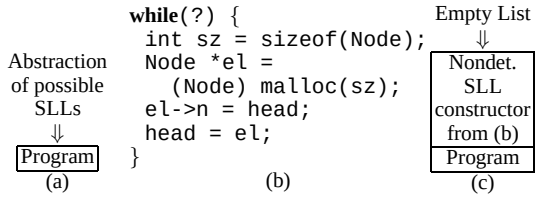


Fig. 10. Illustration of input specifications in TVLA for programs that manipulate singly-linked lists. (a) Traditional input specification in TVLA. (b) A fragment of code that nondeterministically constructs all possible singly-linked lists. (c) The use of loop (b) to specify a set of inputs.

The abstract interpretation of the DSC is performed using an extended vocabulary that contains the new instrumentation relation symbols. The 3-valued structures collected at the exit node of the DSC become the abstract input to the original procedure for the subsequent abstract interpretation of the procedure.

Note that history relations (such as $r_0[n, x](v)$ from §2) are intended to record the state of the store at the entry point to the procedure or, equivalently, at the exit from the DSC. To make sure that these relations have appropriate values, they are maintained in tandem with their active counterparts during abstract interpretation of the DSC. When abstract input refinement is completed, values of history relations are frozen in preparation for the abstract interpretation that is about to be performed on the procedure proper.

The $stable_2$ instrumentation relation of Formula (6) exemplifies the benefits of the DSC methodology. The maintenance of $stable_2$, $t[n]$, $t_0[n]$, and other instrumentation relations starting from the empty store, allows us to conclude that $stable_2$ has value 1 for every tuple of every abstract input structure to procedure `InsertSort` (and so the stability property holds initially).

A DSC is also used to automatically construct the abstract input structures before the first abstract interpretation (see line [1] in Fig. 7). This allows the user to specify the program's inputs in the form of a program, which frees the user from having to know the details of the initial abstraction in use.

[4] In that structure, all history relations, such as $t_0[n]$, have the same values as their active counterparts, but have been omitted from the figure for clarity.

4.4 Success of Refinement for `InsertSort`

In all of the structures collected at the exit node of `InsertSort` by the second run of abstract interpretation, $sorted_1() = 1$. The permutation property also holds on all of the structures. These two facts establish the partial correctness of `InsertSort`. This process required one iteration of abstraction refinement, used a vanilla version of the specification, and needed no user intervention.

5 Experimental Evaluation

To evaluate the techniques presented in this paper, we extended TVLA to perform iterative abstraction refinement, and applied it to three types of queries and five programs (see Fig. 11). Besides `InsertSort`, the test programs included sorting procedures `BubbleSort` and `InsertSort_AS`,[5] list-merging procedure `Merge` and *in-situ* list-reversal procedure `Reverse`.

The antistability query (Formula (7)) asserts that the order of elements with equal `data`-components is reversed. As before, this is only part of the specification of the desired property. It remains to assert that the produced list is a permutation of the input, which is accomplished using Formula (1).

$$\forall v_1, v_2 : (dle(v_1, v_2) \wedge dle(v_2, v_1) \wedge t_0[n](v_1, v_2)) \rightarrow t[n](v_2, v_1). \qquad (7)$$

The DSCs that we used in our tests are programs to generate unsorted lists of arbitrary length, in the case of all programs but `Merge`. For `Merge`, the DSC is a program to generate pairs of unsorted lists.

Test Program	$sorted$	$stable$	$antistable$	$sorted \wedge stable$	$sorted \wedge antistable$
BubbleSort	1	1	$1/2$	1	$1/2$
InsertSort	1	**$1/2$**	$1/2$	**$1/2$**	$1/2$
InsertSort_AS	1	$1/2$	1	$1/2$	1
Merge	$1/2$	1	$1/2$	$1/2$	$1/2$
Reverse	$1/2$	$1/2$	1	$1/2$	$1/2$

Fig. 11. Results from applying iterative abstraction refinement to the verification of properties of programs that manipulate linked lists. Columns 2, 3, and 4 correspond to the queries stated in Formulas (2), (5), and (7), respectively. $sorted \wedge stable$ and $sorted \wedge antistable$ are shorthands for conjunctions of Formulas (2) with (5) and (2) with (7), respectively. The two entries in bold are discussed in §5.1 and §5.2.

Fig. 11 shows that iterative abstraction refinement was able to generate the right instrumentation relations for TVLA to establish several useful facts. For example, TVLA succeeds in demonstrating that all three sorting routines produce sorted lists, that `BubbleSort` and `Merge` are stable routines, and that `InsertSort_AS` and `Reverse` are antistable routines.

Indefinite answers are indicated by $1/2$ entries. *It is important to understand that most of the occurrences of* $1/2$ *in Fig. 11 are the most precise correct answers*. For instance, the result of applying `Reverse` to an unsorted list is usually an unsorted list; however, in the case that the input list happens to be in non-increasing order, `Reverse` produces a sorted list. Consequently, the most precise answer to the query is $1/2$, not 0.

However, the two $1/2$ entries shown in bold in Fig. 11 are not the desired answers: these are the $stable$ and the $sorted \wedge stable$ entries of `InsertSort`. These cases illustrate two reasons why our verification method is sometimes unable to obtain the most precise answer to the query, and are discussed below.

[5] `InsertSort_AS` is identical to `InsertSort` except that it uses $\geq$ instead of $>$ in line [10] of Fig. 5 (i.e., when looking for the correct place to insert the current node). This implementation of insertion sort is antistable.

5.1 Shortcomings of a Subformula-Based Refinement Strategy

Procedure `InsertSort` consists of two nested loops (see Fig. 5). The outer loop traverses the list, setting pointer variable `r` to point to list nodes. For each iteration of the outer loop, the inner loop finds the correct place to insert `r`'s target, by traversing the list from the start using pointer variable `l`; `r`'s target is inserted before `l`'s target when `l->data > r->data`. Because `InsertSort` satisfies the invariant that all list nodes that appear in the list before `r`'s target are already in the correct order, the `data`-component of `r`'s target is less than the `data`-component of all nodes ahead of which `r`'s target is moved. Thus, `InsertSort` preserves the original order of elements with equal `data`-components, and `InsertSort` is a stable routine.

Any approach must be incapable of establishing that some true properties hold for certain programs. This example demonstrates where our present techniques fall short. Our current abstraction-refinement strategy only considers subformulas of the query when it introduces new instrumentation relations. Thus, when verifying the stability of `InsertSort`, only subformulas of Formula (5) are considered. As a result, the sortedness invariant stated above can never be observed, and (when attempting to verify $stable$ alone) TVLA is unable to establish that `InsertSort` is stable.

At present, in such cases, it is possible to let the user suggest additional formulas to be used as the source for new instrumentation relations. Looking to the future, our tool provides a framework in which different strategies for generating instrumentation relations for abstraction refinement can be explored. Our next step in this direction will be to use the weakest-precondition transformation as a method to generate new instrumentation relations.

It is instructive to contrast this example with the verification of antistability of `InsertSort_AS`. When looking for a place to insert `r`'s target, this routine stops when `l->data >= r->data` and inserts `r`'s target before `l`'s target. The analysis need not establish anything about sortedness properties to observe that every list node is inserted before any other node with the same `data`-value. Once the appropriate instrumentation relations are introduced based on subformulas of the antistability query, TVLA is able to establish antistability of `InsertSort_AS`, even in the absence of the sortedness query.

5.2 Imprecision in Abstract Interpretation

When verifying the conjunction of the stability and sortedness queries on `InsertSort`, we allow iterative abstraction refinement to introduce subformulas of both Formulas (2) and (5). As a result, after some iterations, the analysis has established the invariant that nodes appearing in the list before `r`'s target are in correctly sorted order.

Just prior to the insertion of `r`'s target before `l`'s target, this invariant should allow the analysis to establish that the `data`-component of `l`'s target is less than or equal to the `data`-component of all nodes between it and `r`'s target. In addition, because `l->data > r->data`, the `data`-component of `r`'s target is not equal to the `data`-component of those nodes, and the insertion preserves stability. However, the analysis is only able to conclude that the `data`-component of `l`'s target is *possibly* less than or equal to the `data`-component of all nodes between it and `r`'s target. With manual intervention—accomplished in TVLA via the addition of a focus formula or a constraint [20]—it is possible to overcome the imprecision, allowing TVLA to verify the stability of `InsertSort`.

The real source of the problem is that the abstract transformers created by TVLA are only an approximation of the best abstract transformer [7]. Some on-going work is exploring ways to incorporate best transformers into TVLA [22].

5.3 Performance

Fig. 12 gives execution times that were collected on a 3Ghz Linux PC with 4Gb of RAM. The longest-running analysis, which verifies that `BubbleSort` is partially correct, i.e., produces a sorted list, and is stable takes under 6 minutes. The majority of the analyses take less than a minute. These numbers are very close to how long it takes to verify the sortedness query[6] when the user carefully chooses the right instrumentation relations [14]. The maximum amount of memory used by TVLA to perform the analyses varied from just under 2 megabytes to 17.5 megabytes.[7]

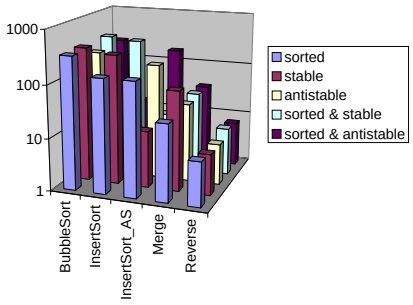


Fig. 12. Execution times in seconds on a log scale.

5.4 Additional Experiments

We performed three additional experiments to test the applicability of our technique to other queries and data structures. The first experiment successfully verified that the *in-situ* list-reversal procedure `Reverse` indeed produces a list that is the reversal of the input list. The query that expresses this property is $\forall\, v_1, v_2 : n(v_1, v_2) \leftrightarrow n_0(v_1, v_2)$. This experiment took only 5 seconds and used less than 2 megabytes of memory.

The second and third experiments involved two programs that manipulate binary-search trees. `InsertBST` inserts a new node into a binary-search tree, and `DeleteBST` deletes a node from a binary-search tree. For both programs our technique successfully verified the query that the nodes of the tree pointed to by variable `t` remain in sorted order at the end of the programs:

$$\forall\, v_1 : r[t](v_1) \rightarrow (\forall\, v_2 : (left(v_1, v_2) \rightarrow dle(v_2, v_1)) \wedge (right(v_1, v_2) \rightarrow dle(v_1, v_2))). \tag{8}$$

The initial specification for the analyses included only three standard instrumentation relations, similar to those listed in Tab. 2. Relation $r[t](v_1)$ from Formula (8), for example, distinguishes nodes in the (sub)tree pointed to by `t`. The DSC used for the analyses non-deterministically constructed a binary-search tree by allocating one new node at a time and inserting it into the tree while maintaining the order property stated in Formula (8). The `InsertBST` experiment took 30 seconds and used less than 3 megabytes of memory, while the `DeleteBST` experiment took approximately 10 minutes and used 37 megabytes of memory.

6 Related Work

The work reported here is most similar in spirit to counterexample-guided abstraction refinement [12, 19, 5, 13, 16, 8, 9, 3, 4, 11]. If a counterexample to the query being verified is found by an analysis equipped with such a technique, the analysis tests whether the counterexample is spurious. If so, new relations that guarantee the elimination of the counterexample are introduced. The analysis is then repeated.

Counterexamples are finite traces through the state graph. The SLAM toolkit [1] implementation uses symbolic execution of the path via the strongest postcondition mechanism to determine whether a counterexample is spurious. If it is, symbolic execution

[6] Sortedness is the only query in our set to which TVLA has been applied before this work.

[7] TVLA is written in Java. Here we report the maximum of total memory minus free memory, as returned by Runtime.

can also be used to identify a new abstraction relation that can eliminate the counterexample. In hardware verification (e.g., [5]), the unabstracted system is finite-state, so the counterexample trace can be tested for spuriousness on the actual system. A spurious counterexample can then be used to minimally refine the system to avoid encountering the same counterexample on a subsequent iteration.

A key difference between our setting and that explored in prior work is the abstraction domain. All abstraction-refinement work to date has used abstract domains that are fixed, finite, Cartesian products of Boolean values. (The use of such domains is known as *predicate abstraction*.) In predicate abstraction, the only relations introduced are nullary relations. Designing an abstraction-refinement technique in which the abstract states are described by 3-valued logical structures, rather than Boolean vectors, calls for a different approach. Our work lifts the predicate-abstraction approach to a more general setting; in particular, the abstraction-refinement algorithm described in this paper will introduce unary, binary, ternary, etc. relations, in addition to nullary relations. This capability is needed for the refinement algorithm to gain the full benefits of the richer class of abstractions that TVLA supports.

In [17], weakest preconditions are used to generate nullary instrumentation relations, which are then generalized manually. The abstract-interpretation technique presented there produces precise results if it terminates, but is not guaranteed to terminate for all cases. In contrast, our method is guaranteed to terminate, and automatically generates interesting non-nullary relations, such as the unary relation $inOrder[dle, n](v)$, which is crucial for showing sortedness and stability.

The concept of a data-structure constructor, which non-deterministically constructs all valid inputs to the program, can be thought of as a mechanism for closing open programs, and hence is related to such work as [6] and [21].

References

1. SLAM toolkit. Available at "http://research.microsoft.com/slam/".
2. TVLA system. Available at "http://www.math.tau.ac.il/∼rumster/TVLA/".
3. T. Ball and S. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, January 2002.
4. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Int. Conf. on Softw. Eng.*, pages 385–395, May 2003.
5. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. Computer-Aided Verif.*, pages 154–169, July 2000.
6. C. Colby, P. Godefroid, and L. Jagadeesan. Automatically closing open reactive programs. In *Conf. on Prog. Lang. Design and Impl.*, pages 345–357, New York, NY, 1998. ACM Press.
7. 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.
8. S. Das and D. Dill. Successive approximation of abstract transition relations. In *Symp. on Logic in Comp. Sci.*, 2001.
9. S. Das and D. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*. Springer, 2002.
10. D. Gopan, F. DiMaio, N.Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Tools and Algs. for the Construct. and Analysis of Syst.*, 2004.
11. T. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proc. Computer-Aided Verif.*, Lec. Notes in Comp. Sci., pages 262–274. Springer, 2003.
12. R. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.
13. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Tools and Algs. for the Construct. and Analysis of Syst.*, Lec. Notes in Comp. Sci., pages 98–112. Springer, 2001.

14. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA 2000: Proc. of the Int. Symp. on Software Testing and Analysis*, pages 26–38, 2000.
15. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.
16. C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible counter-examples when model checking Java programs. In *Tools and Algs. for the Construct. and Analysis of Syst.*, Lec. Notes in Comp. Sci. Springer, 2001.
17. G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Conf. on Prog. Lang. Design and Impl.*, pages 83–94, New York, NY, 2002. ACM Press.
18. T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symp. On Programming*, 2003.
19. V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *Tools and Algs. for the Construct. and Analysis of Syst.*, Lec. Notes in Comp. Sci., pages 178–192. Springer, 1999.
20. 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.
21. O. Tkachuk, M. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Proc. ASE*. Springer, 2003.
22. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *In Proc. TACAS*. Springer, 2004.