# Abstraction Refinement via Inductive Learning

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 how to automatically create abstractions for program analysis. We show that inductive learning, the goal of which is to identify general rules from a set of observed instances, provides new leverage on the problem. An advantage of an approach based on inductive learning is that it does not require the use of a theorem prover.

## 1 Introduction

We present an approach to automatically creating abstractions for use in program analysis. As in some previous work [12, 4, 13, 18, 5, 2, 8], the approach involves the successive refinement of the abstraction in use. Unlike previous work, the work presented in this paper is aimed at programs that manipulate pointers and heap-allocated data structures. However, while we demonstrate our approach on shape-analysis problems, the approach is applicable in any program-analysis setting that uses first-order logic.

The paper presents an abstraction-refinement method for use in static analyses based on 3-valued logic [21], where the semantics of statements and the query of interest are expressed using logical formulas. In this setting, a memory configuration is modeled by 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. Summary individuals 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. The TVLA system is a tool for creating such analyses [1].

With the method proposed in this paper, refinement is performed by introducing new *instrumentation relations* (defined via logical formulas over core relations, which capture the basic properties of memory configurations). Instrumentation relations record auxiliary information in a logical structure, thus providing a mechanism to fine-tune an abstraction: an instrumentation relation captures a property that an individual memory cell may or may not possess. In general, the introduction of additional instrumentation relations refines an abstraction into one that is prepared to track finer distinctions among stores. The choice 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 establishes a new connection between program analysis and machine learning by showing that *inductive logic programming* (ILP) [19, 17, 14] is relevant to the problem of creating abstractions. We use ILP for learning new instrumentation relations that preserve information that would otherwise be lost due to abstraction.

- The method has been implemented as an extension of TVLA. In this system, all of the user-level obligations for which TVLA has been criticized in the past have been addressed. The input required to specify a program analysis consists of: (i) a transition system, (ii) a query (a formula that identifies acceptable outputs), and (iii) a characterization of the program's valid inputs.
- We present experimental evidence of the value of the approach. We tested the method on sortedness, stability, and antistability queries for a set of programs that perform destructive list manipulation, as well as on partial-correctness queries for two binary-search-tree programs. The method succeeds in all cases tested.

Inductive learning concerns identifying general rules from a set of observed instances—in our case, from relationships observed in a logical structure. An advantage of an approach based on inductive learning is that it does not require the use of a theorem prover. This is particularly beneficial in our setting because our logic is undecidable.

The paper is organized as follows: §2 introduces terminology and notation. Readers familiar with TVLA can skip to §2.2, which briefly summarizes ILP. §3 illustrates our goals on the problem of verifying the partial correctness of a sorting routine. §4 describes the techniques used for learning abstractions. (Further details can be found in [16].) §5 presents experimental results. §6 discusses related work.

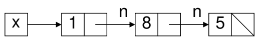



**Fig. 1.** A possible store for a linked list.

## 2 Background

### 2.1 Stores as Logical Structures and their Abstractions

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

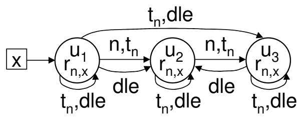
(a)

| 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.

Our work extends the program-analysis framework of [21]. In that approach, concrete memory configurations (i.e., *stores*) are encoded as logical structures 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. 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; in this example, unary relations represent pointer variables and binary relation $n$ represents the n-field of a List cell. The data field is modeled indirectly, via the binary relation *dle* (which stands for "data




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

| $n$ | $u_1$ | $u_2$ | $u_3$ |
|---|---|---|---|
| $u_1$ | 0 | 1 | 0 |
| $u_2$ | 0 | 0 | 1 |
| $u_3$ | 0 | 0 | 0 |

| $t_n$ | $u_1$ | $u_2$ | $u_3$ |
|---|---|---|---|
| $u_1$ | 1 | 1 | 1 |
| $u_2$ | 0 | 1 | 1 |
| $u_3$ | 0 | 0 | 1 |

| *dle* | $u_1$ | $u_2$ | $u_3$ |
|---|---|---|---|
| $u_1$ | 1 | 1 | 1 |
| $u_2$ | 0 | 1 | 0 |
| $u_3$ | 0 | 1 | 1 |

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

less-than-or-equal-to") listed in Tab. 1. Fig. 2 shows 2-valued structure $S_2$, which represents the store of Fig. 1 (relations $t_n$, $r_{n,x}$, and $c_n$ will be explained below).

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 $eq \in \mathcal{R}_2$). A *2-valued logical structure* $S$ over $\mathcal{R}$ is a set of *individuals* $U^S$, along with an *interpretation* that maps each relation symbol $p$ of arity $k$ to a truth-valued function: $p^S\colon (U^S)^k \to \{0,1\}$, where $eq^S$ is the equality relation on individuals. The set of 2-valued structures is denoted by $\mathcal{S}_2[\mathcal{R}]$.

In 3-valued logic, a third truth value—$1/2$—is introduced to denote uncertainty. 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 $l_2 = 1/2$. A *3-valued logical structure* $S$ is defined like a 2-valued logical structure, except that the values in relations can be $\{0, 1/2, 1\}$. An individual for which $eq^S(u,u) = 1/2$ is called a *summary individual*. A summary individual abstracts one or more fragments of a data structure, and can represent more than one concrete memory cell. The set of 3-valued structures is denoted by $\mathcal{S}_3[\mathcal{R}]$.

**Concrete and Abstract Semantics** A concrete operational semantics is defined by specifying a structure transformer for each kind of edge $e$ that can appear in a transition system. A structure transformer is specified by providing *relation-update formulas* for the core relations.[3] These formulas define how the core relations of a 2-valued logical structure $S$ that arises at the source of $e$ are transformed by $e$ to create a 2-valued logical structure $S'$ at the target of $e$. Edge $e$ may optionally have a *precondition formula*, which filters out structures that should not follow the transition along $e$.

However, sets of 2-valued structures do not yield a suitable abstract domain; for instance, when the language being modeled supports allocation from the heap, the set of individuals that may appear in a structure is unbounded, and thus there is no a priori upper bound on the number of 2-valued structures that may arise during the analysis.

To ensure termination, we abstract sets of 2-valued structures using 3-valued structures. A set of stores is then represented by a (finite) set of 3-valued logical structures. The abstraction is defined using an equivalence relation on individuals: 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 (Canonical Abstraction).** Let $S \in \mathcal{S}_2$, and let $\mathcal{A} \subseteq \mathcal{R}_1$ 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 $U^S$:

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

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

If all unary relations are abstraction relations ($\mathcal{A} = \mathcal{R}_1$), the canonical abstraction of 2-valued logical structure $S_2$ is $S_3$, shown in Fig. 3, with $f_{\mathcal{A}}(u_1) = u_1$ and $f_{\mathcal{A}}(u_2) =$

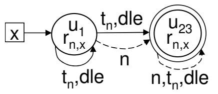
[3] Formulas are first-order formulas with transitive closure: a *formula* over the vocabulary $\mathcal{R}$ is defined as follows (where $p^*(v_1, v_2)$ stands for the reflexive transitive closure of $p(v_1, v_2)$):

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

$f_{\mathcal{A}}(u_3) = u_{23}$. $S_3$ 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. The following graphical notation is used for depicting 3-valued logical structures:

- Individuals are represented by circles containing their names and (non-0) values for unary relations. Summary individuals are represented by double circles.
- A unary relation $p$ corresponding to a pointer-valued program variable is represented by a solid arrow from $p$ to the individual $u$ for which $p(u) = 1$, and by the absence of a $p$-arrow to each node $u'$ for which $p(u') = 0$. (If $p = 0$ for all individuals, the relation name $p$ is not shown.)
- A binary relation $q$ is represented by a solid arrow labeled $q$ between each pair of individuals $u_i$ and $u_j$ for which $q(u_i, u_j) = 1$, and by the absence of a $q$-arrow between pairs $u'_i$ and $u'_j$ for which $q(u'_i, u'_j) = 0$.
- Relations with value $1/2$ are represented by dotted arrows.

Canonical abstraction ensures that each 3-valued structure is no larger than some fixed size, known *a priori*. Moreover, the meaning of a given formula in the concrete domain ($\wp(\mathcal{S}_2)$) is consistent with its meaning in the abstract domain ($\wp(\mathcal{S}_3)$), although the formula's value in an abstract structure $f_{\mathcal{A}}(S)$ may be less precise than its value in the concrete structure $S$.




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

| $n$ | $u_1$ | $u_{23}$ |
|---|---|---|
| $u_1$ | 0 | 1/2 |
| $u_{23}$ | 0 | 1/2 |

| $t_n$ | $u_1$ | $u_{23}$ |
|---|---|---|
| $u_1$ | 1 | 1 |
| $u_{23}$ | 0 | 1/2 |

| $dle$ | $u_1$ | $u_{23}$ |
|---|---|---|
| $u_1$ | 1 | 1 |
| $u_{23}$ | 0 | 1/2 |

**Fig. 3.** A 3-valued structure $S_3$ that is the canonical abstraction of structure $S_2$.

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 collection of equations on variables that take their values in $\wp(\mathcal{S}_3)$ [21].

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

**Table 2.** Defining formulas of some commonly used instrumentation relations. There is a separate reachability relation $r_{n,x}$ for every program variable `x`.

**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 $p \in \mathcal{I}_k \subseteq \mathcal{R}_k$ 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.

The introduction of unary instrumentation relations that are used as abstraction relations provides a way to control which concrete individuals are merged together, and thereby control the amount of information lost by abstraction. Tab. 2 lists some instrumentation relations that are important for the analysis of programs that use type `List`.

## 2.2 Inductive Logic Programming (ILP)

Given a logical structure, the goal of an ILP algorithm is to learn a logical relation (defined in terms of the logical structure's other relations) that agrees with the classification of input examples. ILP algorithms produce the answer in the form of a logic program. (Non-recursive) logic programs correspond to a subset of first-order logic.[4] A logic program can be thought of as a disjunction over the program rules, with each rule corresponding to a conjunction of literals. Variables not appearing in the head of a rule are implicitly existentially quantified.

**Definition (ILP).** Given a set of positive example tuples $E^+$, a set of negative example tuples $E^-$, and a logical structure, the goal of ILP is to find a formula $\psi_E$ such that all $e \in E^+$ are satisfied (or *covered*) by $\psi_E$ and no $e \in E^-$ is satisfied by $\psi_E$. □

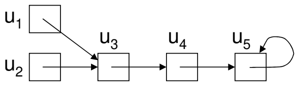



**Fig. 4.** A linked list with shared elements.

For example, consider learning a unary formula that holds for linked-list elements that are pointed to by the `n` fields of more than one element (as used in [11, 3]). We let $E^+ = \{u_3, u_5\}$ and $E^- = \{u_1, u_4\}$ in the 2-valued structure of Fig. 4. The formula $\psi_{\mathit{isShared}}(v) \stackrel{\text{def}}{=} \exists\, v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge \neg eq(v_1, v_2)$ meets the objective, as it covers all positive and no negative example tuples.

Fig. 5 presents the ILP algorithm used by systems such as FOIL [19], modified to construct the answer as a first-order logic formula in disjunctive normal form. This algorithm is capable of learning the formula $\psi_{\mathit{isShared}}(v)$ (by performing one iteration of the outer loop and three iterations of the inner loop to successively choose literals $n(v_1, v)$, $n(v_2, v)$, and $\neg eq(v_1, v_2)$). It is a sequential covering algorithm parameterized by the function $Gain$, which characterizes the usefulness of adding a particular literal (generally, in some heuristic fashion). The algorithm creates a new disjunct as long as there are positive examples that are not covered by existing disjuncts. The disjunct is extended by conjoining a new literal until it covers no negative examples. Each literal uses a relation symbol from the vocabulary of structure $S$; valid arguments to a literal are the variables of target relation $E$, as well as new variables, as long as at least one of the arguments is a variable already used in the current disjunct. In FOIL, one literal is chosen using a heuristic value based on the information gain (see line [7]). FOIL uses information gain to find the literal that differentiates best between positive and negative examples.

`Input:` Target relation $E(v_1, \ldots, v_k)$,
      Structure $S \in \mathcal{S}_3[\mathcal{R}]$,
      Set of tuples $Pos$, Set of tuples $Neg$
[1] $\psi_E := \mathbf{0}$
[2] **while** ($Pos \neq \emptyset$)
[3]   $NewDisjunct := \mathbf{1}$
[4]   $NewNeg := Neg$
[5]   **while** ($NewNeg \neq \emptyset$)
[6]     $Cand :=$ candidate literals using $\mathcal{R}$
[7]     $Best := L \in Cand$ with max $Gain(L, NewDisjunct)$
[8]     $NewDisjunct := NewDisjunct \wedge L$
[9]     $NewNeg :=$ subset of $NewNeg$ satisfying $L$
[10]   $\exists$-quantify $NewDisjunct$ variables $\notin \{v_1, \ldots, v_k\}$
[11]   $\psi_E := \psi_E \vee NewDisjunct$
[12]   $Pos :=$ subset of $Pos$ *not* satisfying $NewDisjunct$

**Fig. 5.** Pseudo-code for FOIL.

# 3 Example: Verifying Sortedness

[4] ILP algorithms are capable of producing recursive programs, which correspond to first-order logic plus a least-fixpoint operator (which is more general than transitive closure).

Given the static-analysis algorithm defined in §2.1, to demonstrate the partial correctness of a procedure, the user must supply the following program-specific information:

- The procedure's control-flow graph.
- A *data-structure constructor* (DSC): a code fragment that non-deterministically constructs all valid inputs.
- A query; i.e., a formula that identifies the intended outputs.

The analysis algorithm is run on the DSC concatenated with the procedure's control-flow graph; the query is then evaluated on the structures that are generated at exit.

```
[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. 6.** A stable version of insertion sort.

Consider the problem of establishing that `InsertSort` shown in Fig. 6 is partially correct. This is an assertion that compares the state of a store at the end of a procedure with its state at the start. In particular, a correct sorting routine must perform a permutation of the input list, i.e. all list elements reachable from variable `x` at the start of the routine must be reachable from `x` at the end. We can express the permutation property as follows:

$$\forall v : r^0_{n,x}(v) \leftrightarrow r_{n,x}(v), \tag{1}$$

where $r^0_{n,x}$ denotes the reachability relation for `x` at the beginning of `InsertSort`. If Formula (1) holds, then the elements reachable from `x` after `InsertSort` executes are exactly the same as those reachable at the beginning, and consequently the procedure performs a permutation of list `x`. In general, for each relation $p$, we have such a *history relation* $p^0$.

Fig. 7 shows the three structures that characterize the valid inputs to `InsertSort` (they represent the set of stores in which program variable `x` points to an acyclic linked list). To verify that `InsertSort` produces a *sorted* permutation of the input list, 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:

$$\forall\, v_1 : r_{n,x}(v_1) \rightarrow (\forall\, v_2 : n(v_1, v_2) \rightarrow \mathit{dle}(v_1, v_2)). \tag{2}$$

If it does, then the nodes reachable from `x` must be in non-decreasing order.

Abstract interpretation collects 3-valued structure $S_3$ shown in Fig. 3 at line [24]. Note that Formula (2) evaluates to $1/2$ on $S_3$. While the first list element is guaranteed to be in correct order with respect to the remaining elements, there is no guarantee that all list nodes represented by the summary node are in correct order. In particular, because $S_3$ represents $S_2$, shown in Fig. 2, the analysis admits the possibility that the (correct) implementation of insertion sort of Fig. 6 can produce the store shown in Fig. 1. Thus, the abstraction that we used was not fine-grained enough to establish the partial correctness of `InsertSort`. In fact, the abstraction is not fine-grained enough to separate the set of sorted lists from the lists not in sorted order.

In [15], Lev-Ami et al. used TVLA to establish the partial correctness of `InsertSort`. The key step was the introduction of instrumentation relation $\mathit{inOrder}_{\mathit{dle},n}(v)$, which holds for nodes whose `data`-components are less than or equal to those of their `n`-successors; $\mathit{inOrder}_{\mathit{dle},n}(v)$ was defined by:

$$\mathit{inOrder}_{\mathit{dle},n}(v) \stackrel{\text{def}}{=} \forall v_1 : n(v, v_1) \rightarrow \mathit{dle}(v, v_1). \qquad (3)$$

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

$$\forall v : r_{n,x}(v) \rightarrow \mathit{inOrder}_{\mathit{dle},n}(v). \qquad (4)$$

After the introduction of relation $\mathit{inOrder}_{\mathit{dle},n}$, the 3-valued structures that are collected by abstract interpretation at the end of `InsertSort` describe all stores in which variable `x` points to an acyclic, *sorted* linked list. In all of these structures, Formulas (4) and (1) evaluate to 1. Consequently, `InsertSort` is guaranteed to work correctly on all valid inputs.

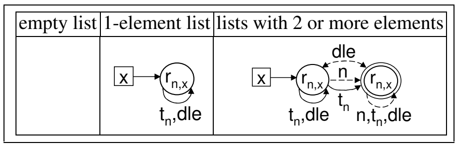



**Fig. 7.** The structures that describe possible inputs to `InsertSort`.

# 4 Learning an Abstraction

In [15], instrumentation relation $\mathit{inOrder}_{\mathit{dle},n}$ 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 this insight into appropriate instrumentation relations.

The goal of this paper is to automate the identification of appropriate instrumentation relations, such as $\mathit{inOrder}_{\mathit{dle},n}$. For `InsertSort`, the goal is to obtain definite answers when evaluating Formula (2) on the structures collected by abstract interpretation at line [24] of Fig. 6. Fig. 8 gives pseudo-code for our method, the steps of which can be explained as follows:

```
Input: a transition system,
       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 rels p_1,...,p_k
[7]    Refine the actions that define
         the transition system
[8]    Refine the abstract input
[9]  while(true)
```

**Fig. 8.** Pseudo-code for iterative abstraction refinement.

- (Line [1]; [16, §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 and §4.2) Find defining formulas for new instrumentation relations.
- (Line [7]) Replace all occurrences of these formulas 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 [20] to generate refined relation-update formulas for the transition system.

– (Line [8]; [16, §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.

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. 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 $1$.

In contrast, when unary instrumentation relation $\mathit{inOrder}_{\mathit{dle},n}$ is present, there are several statements of the program where abstract interpretation results in new definite entries for $\mathit{inOrder}_{\mathit{dle},n}$. For instance, because of the comparison in line [10] of Fig. 6, the insertion in lines [12]–[14] of the node pointed to by `r` (say $u$) before the node pointed to by `l` results in a new definite entry $\mathit{inOrder}_{\mathit{dle},n}(u)$.

An algorithm to generate new instrumentation relations should take into account the sources of imprecision. §4.1 describes subformula-based refinement; §4.2 describes ILP-based refinement. At present, we employ subformula-based refinement first, because the cost of this strategy is reasonable (see §5) and the strategy is often successful. When subformula-based refinement can no longer refine the abstraction, we turn to ILP.

Because a query has finitely many subformulas and we currently limit ourselves to one round of ILP-based refinement, 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.

### 4.1 Subformula-Based Refinement

When the query $\varphi$ evaluates to $1/2$ on a structure $S$ collected at the exit node, we invoke function *instrum*, a recursive-descent procedure to generate defining formulas for new instrumentation relations based on the subformulas of $\varphi$ responsible for the imprecision. The details of function *instrum* are given in [16, §4.1].

**Example.** As we saw in §3, abstract interpretation collects 3-valued structure $S_3$ of Fig. 3 at the exit node of `InsertSort`. The sortedness query (Formula (2)) evaluates to $1/2$ on $S_3$, triggering a call to *instrum* with Formula (2) and structure $S_3$, as arguments. Column 2 of Tab. 3 shows the instrumentation relations that are created as a result of the call. Note that $sorted_3$ is defined exactly as $\mathit{inOrder}_{\mathit{dle},n}$, which was the key insight for the results of [15]. □

| $p$ | $\psi_p$ (after call to *instrum*) | $\psi_p$ (final version) |
|---|---|---|
| $\mathit{sorted}_1()$ | $\forall v_1 : r_{n,x}(v_1) \to (\forall v_2 : n(v_1, v_2) \to \mathit{dle}(v_1, v_2))$ | $\forall v_1 : \mathit{sorted}_2(v_1)$ |
| $\mathit{sorted}_2(v_1)$ | $r_{n,x}(v_1) \to (\forall v_2 : n(v_1, v_2) \to \mathit{dle}(v_1, v_2))$ | $r_{n,x}(v_1) \to \mathit{sorted}_3(v_1)$ |
| $\mathit{sorted}_3(v_1)$ | $\forall v_2 : n(v_1, v_2) \to \mathit{dle}(v_1, v_2)$ | $\forall v_2 : \mathit{sorted}_4(v_1, v_2)$ |
| $\mathit{sorted}_4(v_1, v_2)$ | $n(v_1, v_2) \to \mathit{dle}(v_1, v_2)$ | $n(v_1, v_2) \to \mathit{dle}(v_1, v_2)$ |

**Table 3.** Instrumentation relations created by subformula-based refinement.

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 accomplish this, refinement of the program's actions (line [7] in Fig. 8) replaces all occurrences of the defining formulas for the new instrumentation relations in the query and in the definitions of other instrumentation relations with the use of the corresponding new instrumentation-relation symbols.

**Example.** For `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 $\psi_{sorted_1}, \ldots, \psi_{sorted_4}$. These occurrences are replaced with the names of the four relations. In this case, only the new relations' definitions are changed, yielding the definitions given in Column 3 of Tab. 3.

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 the basic version of the specification (the vocabulary consisted of the relations of Tabs. 1 and 2, together with the corresponding history relations), and needed no user intervention. □

### 4.2 ILP-Based Refinement

**Shortcomings of Subformula-Based Refinement** To illustrate a weakness in subformula-based refinement, we introduce the stability property. The stability 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.[5]

$$\forall v_1, v_2 : (\mathit{dle}(v_1, v_2) \wedge \mathit{dle}(v_2, v_1) \wedge t_n^0(v_1, v_2)) \rightarrow t_n(v_1, v_2) \tag{5}$$

Procedure `InsertSort` consists of two nested loops (see Fig. 6). 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.

However, subformula-based refinement is not capable of establishing the stability of `InsertSort`. By considering only subformulas of the query (in this case, Formula (5)) as candidate instrumentation relations, the strategy is unable to introduce instrumentation relations that maintain information about the *transitive* successors with which a list node has the correct relative order.

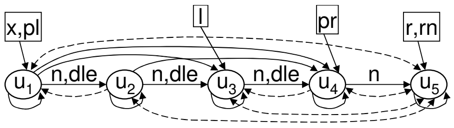
**Learning Instrumentation Relations** Fig. 9 shows the structure $S_9$, which arises during abstract interpretation just before line [6] of Fig. 6, together with a tabular version of relations $t_n$ and *dle*. (We omit reachability relations from the figure for clarity.) After the assignment `l = x;`, nodes $u_2$ and $u_3$ have identical vectors of values for the unary abstraction relations. The subsequent application of canonical abstraction produces structure $S_{10}$, shown in Fig. 10. Bold entries of tables in Fig. 9 indicate definite

[5] A related property, antistability, asserts that the order of elements with equal `data`-components is reversed: $\forall v_1, v_2 : (\mathit{dle}(v_1, v_2) \wedge \mathit{dle}(v_2, v_1) \wedge t_n^0(v_1, v_2)) \rightarrow t_n(v_2, v_1)$
Our test suite also includes program `InsertSort_AS`, which is identical to `InsertSort` except that it uses $\geq$ instead of $>$ in line [10] of Fig. 6 (i.e., when looking for the correct place to insert the current node). This implementation of insertion sort is antistable.

values that are transformed into $1/2$ in $S_{10}$. Structure $S_9$ satisfies the sortedness invariant discussed above: every node among $u_1, ..., u_4$ has the *dle* relationship with all nodes appearing later in the list, except `r`'s target, $u_5$. However, a piece of this information is lost in structure $S_{10}$: *dle*$(u_{23}, u_{23}) = 1/2$, indicating that some nodes represented by summary node $u_{23}$ might not be in sorted order with respect to their successors. We will refer to such abstraction steps as *information-loss points*.

An abstract structure transformer may temporarily create a structure $S_1$ that is not in the image of canonical abstraction [21]. The subsequent application of canonical abstraction transforms $S_1$ into structure $S_2$ by grouping a set $U_1$ of two or more individuals of $S_1$ into a single summary individual of $S_2$. The loss of precision is due to one or both of the following circumstances:

- One of the individuals in $U_1$ possesses a property that another individual does not possess; thus, the property for the summary individual is $1/2$.
- Individuals in $U_1$ have a property in common, which cannot be recomputed precisely in $S_2$.

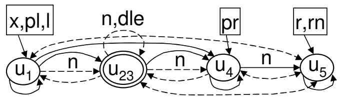



| *dle* | $u_1$ | $u_2$ | $u_3$ | $u_4$ | $u_5$ |
|---|---|---|---|---|---|
| $u_1$ | 1 | 1 | 1 | 1 | 1/2 |
| $u_2$ | 1/2 | **1** | **1** | 1 | 1/2 |
| $u_3$ | **0** | **0** | **1** | 1 | 1/2 |
| $u_4$ | 0 | **0** | 1/2 | 1 | 1/2 |
| $u_5$ | 1/2 | 1/2 | 1/2 | 1/2 | 1 |

| $t_n$ | $u_1$ | $u_2$ | $u_3$ | $u_4$ | $u_5$ |
|---|---|---|---|---|---|
| $u_1$ | 1 | 1 | 1 | 1 | 1 |
| $u_2$ | 0 | **1** | **1** | 1 | 1 |
| $u_3$ | 0 | **0** | **1** | 1 | 1 |
| $u_4$ | 0 | 0 | 0 | 1 | 1 |
| $u_5$ | 0 | 0 | 0 | 0 | 1 |

**Fig. 9.** Structure $S_9$, which arises just before line [6] of Fig. 6. Unlabeled edges between nodes represent the *dle* relation.

In both cases, the solution lies in the introduction of new instrumentation relations. In the former case, it is necessary to introduce a unary abstraction relation to keep the individuals of $U_1$ that possess the property from being grouped with those that do not. In the latter case, it is sufficient to introduce a non-abstraction relation of appropriate arity that captures the common property of individuals in $U_1$. The algorithm described in §2.2 can be used to learn formulas for the following three kinds of relations:[6]




| *dle* | $u_1$ | $u_{23}$ | $u_4$ | $u_5$ |
|---|---|---|---|---|
| $u_1$ | 1 | 1 | 1 | 1/2 |
| $u_{23}$ | **1/2** | **1/2** | 1 | 1/2 |
| $u_4$ | 0 | **1/2** | 1 | 1/2 |
| $u_5$ | 1/2 | 1/2 | 1/2 | 1 |

| $t_n$ | $u_1$ | $u_{23}$ | $u_4$ | $u_5$ |
|---|---|---|---|---|
| $u_1$ | 1 | 1 | 1 | 1 |
| $u_{23}$ | 0 | **1/2** | 1 | 1 |
| $u_4$ | 0 | 0 | 1 | 1 |
| $u_5$ | 0 | 0 | 0 | 1 |

**Fig. 10.** Structure $S_{10}$, corresponding to the transformation of $S_9$ by the statement on line [6] of Fig. 6. Unlabeled edges between nodes represent the *dle* relation.

**Type I:** Unary relation $r_1$ with $E^+ = \{u\}$ for one $u \in U_1$, and $E^- = U_1 - \{u\}$.
**Type II:** Unary relation $r_2$ with $E^+ = U_1$.
**Type III:** Binary relation $r_3$ with $E^+ = U_1 \times U_1$.

Type I relations are intended to prevent the grouping of individuals with different properties, while Types II and III are intended to capture the common properties of individuals in $U_1$. (Type III relations can be generalized to higher-arity relations.)

For the logical structure that serves as input to ILP, we pass the structure $S_1$ identified at an information-loss point. We restrict the algorithm to use only non-history

[6] These are what are needed for our analysis framework, which uses abstractions that generalize predicate-abstraction domains. A fourth use of ILP provides a new technique for predicate abstraction itself: ILP can be used to identify nullary relations that differentiate a positive-example structure $S$ from the other structures arising at a program point. The steps of ILP go beyond merely forming Boolean combinations of existing relations; they involve the creation of new relations by introducing quantifiers during the learning process.

relations of the structure that lose definite entries as a result of abstraction (e.g., $t_n$ and *dle* in the above example). Definite entries of those relations are then used to learn formulas that evaluate to 1 for every positive example and to 0 for every negative example.

We modified the algorithm of §2.2 to learn multiple formulas in one invocation of the algorithm. Our motivation is not to find a single instrumentation relation that explains something about the structure, but rather to find all instrumentation relations that help the analysis establish the property of interest. Whenever we find multiple literals of the same quality (see line [7] of Fig. 5), we extend distinct copies of the current disjunct using each of the literals, and then we extend distinct copies of the current formula using the resulting disjuncts.

This variant of ILP is able to learn a useful binary formula using structure $S_9$ of Fig. 9. The set of individuals of $S_9$ that are grouped by the abstraction is $U = \{u_2, u_3\}$, so the input set of positive examples is $\{(u_2, u_2), (u_2, u_3), (u_3, u_2), (u_3, u_3)\}$. The set of relations that lose definite values due to abstraction includes $t_n$ and *dle*. Literal $\mathit{dle}(v_1, v_2)$ covers three of the four examples because it holds for bindings $(v_1, v_2) \mapsto (u_2, u_2), (v_1, v_2) \mapsto (u_2, u_3)$, and $(v_1, v_2) \mapsto (u_3, u_3)$. The algorithm picks that literal and, because there are no negative examples, $\mathit{dle}(v_1, v_2)$ becomes the first disjunct. Literal $\neg t_n(v_1, v_2)$ covers the remaining positive example, $(u_3, u_2)$, and the algorithm returns the formula

$$\psi_{r_3}(v_1, v_2) \stackrel{\text{def}}{=} \mathit{dle}(v_1, v_2) \lor \neg t_n(v_1, v_2), \tag{6}$$

which can be re-written as $t_n(v_1, v_2) \rightarrow \mathit{dle}(v_1, v_2)$.

Relation $r_3$ allows the abstraction to maintain information about the transitive successors with which a list node has the correct relative order. In particular, although $\mathit{dle}(u_{23}, u_{23})$ is $1/2$ in $S_{10}$, $r_3(u_{23}, u_{23})$ is 1, which allows establishing the fact that all list nodes appearing prior to `r`'s target are in sorted order.

Other formulas, such as $\mathit{dle}(v_1, v_2) \lor t_n(v_2, v_1)$, are also learned using ILP (cf. Fig. 12). Not all of them are useful to the verification process, but introducing extra instrumentation relations cannot harm the analysis, aside from increasing its cost.

## 5 Experimental Evaluation

We extended TVLA to perform iterative abstraction refinement, and applied it to three queries and five programs (see Fig. 11). Besides `InsertSort`, the test programs included sorting procedures `BubbleSort` and `InsertSort_AS`, list-merging procedure `Merge`, and *in-situ* list-reversal procedure `Reverse`.

At present, we employ subformula-based refinement first. During each iteration of subformula-based refinement, we save logical structures at information-loss points. Upon the failure of subformula-based refinement, we invoke the ILP algorithm described in §4.2. To lower the cost of the analysis we prune the returned set of formulas. For example, we currently remove formulas defined in terms of a single relation symbol; such formulas are usually tautologies (e.g., $\mathit{dle}(v_1, v_2) \lor \mathit{dle}(v_2, v_1)$). We then define new instrumentation relations, and use these relations to refine the abstraction by performing the steps of lines [7] and [8] of Fig. 8. Our implementation can learn relations of all types described in §4.2: unary, binary, as well as nullary. However, due to the present cost of maintaining many unary instrumentation relations in TVLA, in the experiments reported here we only learn binary formulas (i.e., of Type III). Moreover, we define new instrumentation relations using only learned formulas of a simple form (currently, those with two atomic subformulas). We are in the process of extending our techniques for pruning useless instrumentation relations. This should make it practical for us to use all types of relations that can be learned by ILP for refining the abstraction.

**Example.**When attempting to verify the stability of `InsertSort`, ILP creates nine formulas including Formula (6). The subsequent run of the analysis successfully verifies the stability of `InsertSort`. □

| Test Program | $sorted$ | $stable$ | $antistable$ |
|---|---|---|---|
| BubbleSort | 1 | 1 | $1/2$ |
| InsertSort | 1 | 1 | $1/2$ |
| InsertSort_AS | 1 | $1/2$ | 1 |
| Merge | $1/2$ | 1 | $1/2$ |
| Reverse | $1/2$ | $1/2$ | 1 |

**Fig. 11.** Results from applying iterative abstraction refinement to the verification of properties of programs that manipulate linked lists.

Fig. 11 shows that the method was able to generate the right instrumentation relations for TVLA to establish all properties that we expect to hold. Namely, TVLA succeeds in demonstrating that all three sorting routines produce sorted lists, that `BubbleSort`, `InsertSort`, 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 all 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 $0$.

| Test Program | $sorted$ # instrum rels total/ILP | $stable$ # instrum rels total/ILP | $antistable$ # instrum rels total/ILP |
|---|---|---|---|
| BubbleSort | $31/0$ | $32/0$ | $41/9$ |
| InsertSort | $39/0$ | $49/9$ | $43/3$ |
| InsertSort_AS | $39/0$ | $43/3$ | $40/0$ |
| Merge | $30/3$ | $28/0$ | $31/3$ |
| Reverse | $26/3$ | $27/3$ | $24/0$ |

**Fig. 12.** The numbers of instrumentation relations (total and learned by ILP) used during the last iteration of abstraction refinement.

Fig. 12 shows the numbers of instrumentation relations used during the last iteration of abstraction refinement. The number of ILP-learned relations used by the analysis is small relative to the total number of instrumentation relations.

Fig. 13 gives execution times that were collected on a 3GHz Linux PC. The longest-running analysis, which verifies that `InsertSort` is stable, takes 8.5 minutes. Seven of the analyses take under a minute. The rest take between 70 seconds and 6 minutes. The total time for the 15 tests is 35 minutes. These numbers are very close to how long it takes to verify the sortedness queries when the user carefully chooses the right instrumentation relations [15].[7] The maximum amount of memory used by the analyses varied from just under 2 MB to 32 MB.[8]

The cost of the invocations of the ILP algorithm when attempting to verify the antistability of `BubbleSort` was 25 seconds (total, for 133 information-loss points). For all other benchmarks, the ILP cost was less than ten seconds.

Three additional experiments tested the applicability of our method to other queries and data structures. In the first experiment, subformula-based refinement 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 $\forall v_1, v_2 : n(v_1, v_2) \leftrightarrow n^0(v_2, v_1)$. This experiment took only 5 seconds and used less than 2 MB of memory. The second and third experiments involved two programs that manipu-

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

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

late 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, subformula-based refinement 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:

$$\forall v_1\colon r_t(v_1) \rightarrow (\forall v_2\colon (\mathit{left}(v_1, v_2) \rightarrow \mathit{dle}(v_2, v_1)) \wedge (\mathit{right}(v_1, v_2) \rightarrow \mathit{dle}(v_1, v_2))) \quad (7)$$

The initial specifications for the analyses included only three standard instrumentation relations, similar to those listed in Tab. 2. Relation $r_t(v_1)$ from Formula (7), for example, distinguishes nodes in the (sub)tree pointed to by `t`. The `InsertBST` experiment took 30 seconds and used less than 3 MB of memory, while the `DeleteBST` experiment took approximately 10 minutes and used 37 MB of memory.

## 6 Related Work

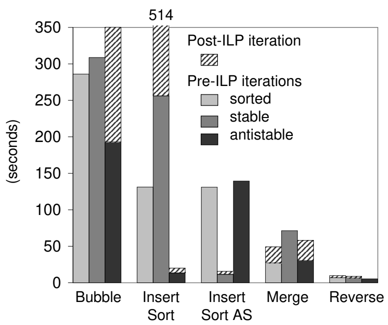



**Fig. 13.** Execution times. For each program, the three bars represent the sorted, stable, and antistable queries. In cases where subformula-based refinement failed, the upper portion of the bars shows the cost of the last iteration of the analysis (on both the DSC and the program) together with the ILP cost.

The work reported here is similar in spirit to counterexample-guided abstraction refinement [12, 4, 13, 18, 5, 2, 8, 6]. A key difference between this work and prior work in the model-checking community is the abstract domain: prior work has used abstract domains that are fixed, finite, Cartesian products of Boolean values (i.e., predicate-abstraction domains), and hence the only relations introduced are nullary relations. Our work applies to a richer class of abstractions—3-valued structures—that generalize predicate-abstraction domains. The abstraction-refinement algorithm described in this paper can introduce unary, binary, ternary, etc. relations, in addition to nullary relations. While we demonstrated our approach using shape-analysis queries, this approach is applicable in any setting in which first-order logic is used to describe program states.

A second distinguishing feature of our work is that the method is driven not by counterexample traces, but instead by imprecise results of evaluating a query (in the case of subformula-based refinement) and by loss of information during abstraction steps (in the case of ILP-based refinement). There do not currently exist theorem provers for first-order logic extended with transitive closure capable of identifying infeasible error traces [9]; hence we needed to develop techniques different from those used in SLAM, BLAST, etc. SLAM identifies the shortest prefix of a spurious counterexample trace that cannot be extended to a feasible path; in general, however, the first information-loss point occurs before the end of the prefix. Information-loss-guided refinement can identify the earliest points at which information is lost due to abstraction, as well as what new instrumentation relations need to be added to the abstraction at those points. A potential advantage of counterexample-guided refinement over information-loss-guided refinement is that the former is goal-driven. Information-loss-guided refinement can discover many relationships that do not help in establishing the query. To alleviate this problem, we restricted the ILP algorithm to only use relations that occur in the query.

Abstraction-refinement techniques from the abstract-interpretation community are capable of refining domains that are not based on predicate abstraction. In [10], for example, a polyhedra-based domain is dynamically refined. Our work is based on a different abstract domain, and led us to develop some new approaches to abstraction refinement, based on machine learning.

In the abstract-interpretation community, a strong (albeit often unattainable) form of abstraction refinement has been identified in which the goal is to make abstract interpretation complete (a.k.a. "optimal") [7]. In our case, the goal is to extend the abstraction just enough to be able to answer the query, rather than to make the abstraction optimal.

## References


1. TVLA system. http://www.cs.tau.ac.il/ tvla/.
2. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, pages 103–122, 2001.
3. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, 1990.
4. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.
5. S. Das and D. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD*, pages 19–32, 2002.
6. C. Flanagan. Software model checking via iterative abstraction refinement of constraint logic queries. In *CP+CV*, 2004.
7. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.
8. T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
9. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive closure logics. In *CSL*, pages 160–174, 2004.
10. B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *SAS*, pages 39–50, 1999.
11. N. Jones and S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, 1981.
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 *TACAS*, pages 98–112, 2001.
14. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.
15. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA*, pages 26–38, 2000.
16. A. Loginov, T. Reps, and M. Sagiv. Learning abstractions for verifying data-structure properties. report TR-1519, Comp. Sci. Dept., Univ. of Wisconsin, January 2005. Available at "http://www.cs.wisc.edu/wpis/papers/tr1519.ps".
17. S. Muggleton. Inductive logic programming. *New Generation Comp.*, 8(4):295–317, 1991.
18. C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible counter-examples when model checking Java programs. In *TACAS*, pages 284–298, 2001.
19. J.R. Quinlan. Learning logical definitions from relations. *Mach. Learn.*, 5:239–266, 1990.
20. T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas with applications to program analysis. In *ESOP*, pages 380–398, 2003.
21. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.