

Learning Abstractions for Verifying Data-Structure Properties

Alexey Loginov
Univ. of Wisconsin
alexey@cs.wisc.edu

Thomas Reps
Univ. of Wisconsin
reps@cs.wisc.edu

Mooly Sagiv
Tel Aviv Univ.
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. We present two strategies for refining an abstraction. The simpler strategy is effective in many cases. The second strategy uses a known machine-learning algorithm to learn an appropriate abstraction. A tool that incorporates the method has been implemented and applied to several programs that manipulate linked lists and binary-search trees. In all cases, the tool is able to demonstrate (i) the partial correctness of the programs, and (ii) that the programs possess additional properties—e.g., stability or antistability.

1. INTRODUCTION

This paper presents an approach to automatically creating abstractions for use in program analysis. As in some previous work [12, 5, 13, 21, 7, 3, 9], the approach 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 [27], 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). Our abstraction-refinement method uses two refinement strategies. The first strategy, *subformula-based refinement*, analyzes the sources of imprecision in the evaluation of the query, and chooses how to define new instrumentation relations using subformulas of the query. The second strategy, *ILP-based refinement*, employs *inductive logic programming* [22, 19, 20, 14, 23], [18, §10], a machine-learning technique, to learn new instrumentation relations that can stave off imprecision due to abstraction.

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 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 [16, 1].

Summary individuals play a crucial role. They are used to ensure that abstract descriptors have an *a priori* bounded size, which guarantees that a fixed-point is always reached. However, the constraint of working with limited-size descriptors implies a loss of information about the store. Intuitively, 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 [27, Theorem 4.9]. 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, the introduction of additional instrumentation relations refines an abstraction into one that is prepared to track finer distinctions among stores. This allows more properties of the program's stores to be identified.

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 is a step towards automatically generating useful abstractions for static analyses based on 3-valued logic.
- We introduce the use of *inductive logic programming* (ILP) for learning new instrumentation relations that preserve information that would otherwise be lost due to abstraction.
- 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, as a transition system)
 - a query; i.e., a formula that identifies acceptable outputs
 - a characterization of the program's allowable inputs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

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 ?

Table 1: Core relations used for representing the stores manipulated by programs that use type `List`.

- The method 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 tested the effectiveness of the method using sortedness, stability, and antistability queries on a collection of programs that perform destructive list manipulation, as well as by using it to establish partial correctness of two binary-search-tree programs. The method is successful in all cases tested here.

The remainder of 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 subformula-based refinement and illustrates it on the example of §3. §5 discusses a shortcoming of subformula-based refinement and describes ILP-based refinement, which uses ILP for learning an abstraction. §6 presents experimental results. §7 discusses related work.

2. BACKGROUND

2.1 Stores as Logical Structures

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

Figure 1: Declaration of a linked-list datatype.

This section summarizes the shape-analysis framework described in [27]. 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 lists the relations that would be used to represent the stores manipulated by programs that use type `List` (declared as shown in Fig. 1), such as the store in Fig. 2. 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. Fig. 3 shows 2-valued structure S_3 , which represents the store of Fig. 2 (relations t_n , $r_{n,x}$, and c_n are explained in §2.1.2).

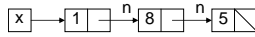


Figure 2: A possible store for a linked list.

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

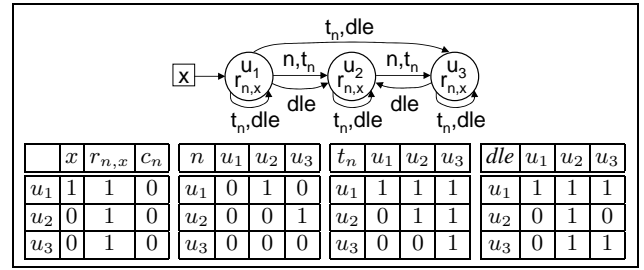


Figure 3: A logical structure S_3 that represents the store shown in Fig. 2 in graphical and tabular representations.

imposes the constraint that x can hold for at most one individual in any structure.

Let $\mathcal{R} = \{eq, p_1, \dots, p_n\}$ 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 $S_2[\mathcal{R}]$.

DEFINITION 1. 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: $\iota^S(p): (U^S)^k \rightarrow \{0, 1\}$. 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, $\iota^S(eq)(u_1, u_2) = 0$.

In 3-valued logic, a third truth value— $1/2$ —is introduced to denote uncertainty: 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 $l_2 = 1/2$. Thus, $1/2 \sqcup 0 = 1/2 \sqcup 1 = 0 \sqcup 1 = 1/2$.

DEFINITION 2. 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: $\iota^S(p): (U^S)^k \rightarrow \{0, 1/2, 1\}$. In addition, (i) for all $u \in U^S$, $\iota^S(eq)(u, u) \sqsupseteq 1$, and (ii) for all $u_1, u_2 \in U^S$ such that u_1 and u_2 are distinct individuals, $\iota^S(eq)(u_1, u_2) = 0$. The set of 3-valued structures is denoted by $S_3[\mathcal{R}]$.

An individual for which $\iota^S(eq)(u, u) = 1/2$ 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.

2.1.1 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 CFG. A structure transformer is specified by providing a collection of *relation-update formulas*, $c(v_1, \dots, v_k) = \tau_{c,e}(v_1, \dots, v_k)$, one for each core relation c .¹ 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

¹Formulas are first-order formulas with transitive closure: a *formula* over the vocabulary $\mathcal{R} = \{eq, p_1, \dots, p_n\}$ is defined as follows (where $p^*(v_1, v_2)$ stands for the reflexive transitive closure of $p(v_1, v_2)$):

$$\begin{aligned}
 p \in \mathcal{R}, \quad \varphi & ::= \mathbf{0} \mid \mathbf{1} \mid p(v_1, \dots, v_k) \mid (\neg \varphi_1) \mid (v_1 = v_2) \\
 \varphi \in \text{Formulas}, & \quad \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 \vee \varphi_2) \mid (\varphi_1 \rightarrow \varphi_2) \\
 \varphi \in \text{Variables} & \quad \mid (\exists v: \varphi_1) \mid (\forall v: \varphi_1) \mid p^*(v_1, v_2)
 \end{aligned}$$

An *assignment* Z is a function that maps variables to individuals (i.e., it has the functionality $Z: \{v_1, v_2, \dots\} \rightarrow U^S$).

For an assignment Z , the (2-valued) *meaning* of a formula φ , denoted by $\llbracket \varphi \rrbracket_2^S(Z)$, yields a truth value in $\{0, 1\}$; the (3-valued) *meaning* of φ , denoted by $\llbracket \varphi \rrbracket_3^S(Z)$, yields a truth value in $\{0, 1/2, 1\}$. $\llbracket \varphi \rrbracket_2^S(Z)$ and $\llbracket \varphi \rrbracket_3^S(Z)$ are defined inductively in the standard fashion (e.g., see [27]).

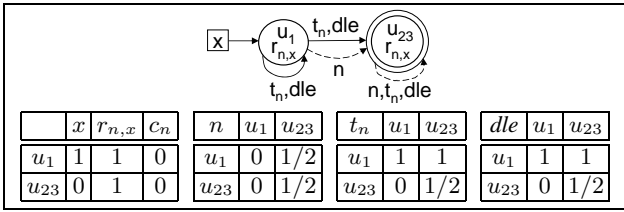


Figure 4: A 3-valued structure S_4 that is the canonical abstraction of structure S_3 .

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 $\wp(S_2) \rightarrow \wp(S_2)$. However, $\wp(S_2)$ is not suitable as an 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 cardinality of elements of $\wp(S_2)$.

One can sidestep this problem by abstracting sets of 2-valued structures using 3-valued structures equipped with a suitable order [27]. 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, 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 3 (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 :*

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

and the surjective function $f_{\mathcal{A}} : U \rightarrow U / \simeq_{\mathcal{A}}$, 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_3 is S_4 , shown in Fig. 4, with $f_{\mathcal{A}}(u_1) = u_1$ and $f_{\mathcal{A}}(u_2) = f_{\mathcal{A}}(u_3) = u_{23}$. S_4 represents all lists with two or more elements, in which the first element's `data` value is no higher 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 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-valued 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 $\iota(q)(u'_i, u'_j) = 0$.
- Relations with value 1/2 are represented by dotted arrows.

p	Intended Meaning	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?	$\exists v_1 : x(v_1) \wedge t_n(v_1, v)$
$c_n(v)$	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 relation $r_{n,x}$ for every program variable x .

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, $\wp(S_2)$) and the abstract domain ($\wp(S_3)$). Thanks to the Embedding Theorem [27, Theorem 4.9], the meaning of the two interpretations is consistent with respect to the abstraction, although the value of a formula on an abstract structure $f_{\mathcal{A}}(S)$ 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 can be 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 $\wp(S_3)$ [27].

2.1.2 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 $p \in \mathcal{I}_k \subseteq \mathcal{R}_k$ is defined by an *instrumentation-relation definition formula* $\psi_p(v_1, \dots, v_k)$. 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 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 the $*$ operator, 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. 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:

$$\forall v : r_{n,x}^0(v) \leftrightarrow r_{n,x}(v). \quad (1)$$

If Formula (1) 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 $p(v_1, \dots, v_k)$, the relation $same_p()$ is defined by $\psi_{same_p} = \forall v_1, \dots, v_k : p(v_1, \dots, v_k) \leftrightarrow p^0(v_1, \dots, v_k)$. 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 address this problem, after a transition from structure S to S' via transformer τ , the new value for an instrumentation relation p can be computed incrementally from the known value of p in S . An algorithm that uses τ and p ’s defining formula $\psi_p(v_1, \dots, v_k)$ to generate an appropriate incremental *relation-maintenance formula* $\mu_{p,\tau}$ is given in [25].

2.2 Inductive Logic Programming (ILP)

ILP is a subfield of Machine Learning, which is itself a subfield of Artificial Intelligence. The goal of an ILP algorithm is to learn a logical relation that agrees with the classification of input examples, given background knowledge. ILP algorithms produce the answer in the form of a logic program. (Non-recursive) logic programs correspond to a subset of first-order logic.² 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 4 (ILP). *Given a set of positive example tuples E^+ , a set of negative example tuples E^- , and a set of background facts B (represented by definite entries of a logical structure), the goal of ILP is to find a formula ψ_E such that all $e \in E^+$ are satisfied (or covered) by ψ_E and no $e \in E^-$ is satisfied by ψ_E .*

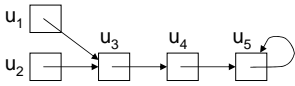


Figure 5: A linked list with shared elements.

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. (The importance of the concept of sharing in heap data structures was recognized in [11, 4]). We let $E^+ = \{u_3, u_5\}$, $E^- = \{u_1, u_4\}$, and $B =$ the 2-valued structure of Fig. 5. The formula $\psi_{isShared}(v) \stackrel{\text{def}}{=} \exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge \neg eq(v_1, v_2)$ meets the objective, as it covers all positive and no negative example tuples.

Fig. 6 shows the simplified ILP algorithm used by systems such as FOIL [22], modified to construct the answer as a first-order logic formula in disjunctive normal form. This algorithm is capable of learning the formula $\psi_{isShared}(v)$. 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 background structure’s vocabulary; valid arguments to a literal are the variables of target relation E , as well as new variables, as long

²ILP 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).

```

Input: Target relation  $E(v_1, \dots, v_k)$ ,
       Structure  $S \in \mathcal{S}_3[\mathcal{R}]$ ,
       Set of tuples  $Pos$ , Set of tuples  $Neg$ 
[1]  $\psi_{target} := \mathbf{0}$ 
[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  $\notin \{v_1, \dots, v_k\}$ 
[11]     $\psi_{target} := \psi_{target} \vee NewDisjunct$ 
[12]     $Pos :=$  subset of  $Pos$  not satisfying  $NewDisjunct$ 

```

Figure 6: Pseudo-code for FOIL.

as at least one of the arguments is a variable already used in the current disjunct. In FOIL, one literal is chosen (see line [7]) using a heuristic value based on the information gain. The reader is referred to [22] for more details.

3. EXAMPLE: VERIFYING SORTEDNESS

```

[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] }

```

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 set of 3-valued structures that identify acceptable inputs.
- A query; i.e., a formula that identifies 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.

Figure 7: A stable version of insertion sort.

Consider the problem of establishing that the version of `InsertSort` shown in Fig. 7 is partially correct. Fig. 8 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:

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

If the formula evaluates to 1, then the nodes reachable from x must be in non-decreasing order.³

³A second property required of a correct sorting procedure (as well as of many other procedures that manipulate linked lists) is that the output list must be a permutation of the input list. This can be established by also checking Formula (1) from §2.

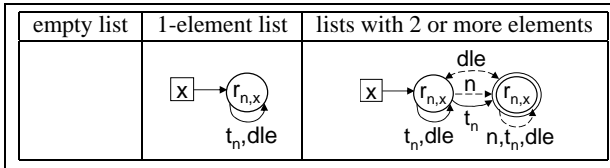


Figure 8: The structures that describe possible inputs to InsertSort.

Fig. 3 shows 2-valued structure S_3 , in which the middle list node must have a larger data-value than the other two nodes. (One of the stores that this structure represents is shown in Fig. 2.) Fig. 7 is a correct implementation of insertion sort, and hence S_3 does not represent any store that can arise at line [24]. Given the structures shown in Fig. 8 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. For instance, S_4 represents the store of Fig. 2. 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 $inOrder_{dle,n}(v)$, 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:

$$inOrder_{dle,n}(v) \stackrel{\text{def}}{=} \forall v_1 : n(v, v_1) \rightarrow dle(v, v_1). \quad (3)$$

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

$$\forall v : r_{n,x}(v) \rightarrow inOrder_{dle,n}(v). \quad (4)$$

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 stores in which variable *x* points to an acyclic, *sorted* linked list. In all of these 3-valued structures, Formulas (4) and (1) evaluate to 1. Consequently, InsertSort is guaranteed to work correctly on all acceptable inputs.

4. LEARNING AN ABSTRACTION

In [15], 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 this insight into appropriate instrumentation relations (e.g., Formula (3)).

The goal of the present paper is to automate the identification of appropriate 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. 7. Fig. 9 gives pseudo-code for our method, 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.

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  $S_1, \dots, S_k$  be the set of
       3-valued structures at exit
[5]   if for all  $S_i$ ,  $\llbracket \varphi \rrbracket_3^i(\llbracket \cdot \rrbracket) \neq 1/2$  break
[6]   Find formulas  $\psi_{p_1}, \dots, \psi_{p_k}$  for new
       instrumentation relations  $p_1, \dots, p_k$ 
[7]   Refine the actions that define
       the program’s transition relation
[8]   Refine the abstract input
[9]   while(true)
```

Figure 9: Pseudo-code for iterative abstraction refinement.

- (Line [6]; §4.1 and §5) Identify formulas to be used to define new instrumentation relations.
- (Line [7]; §4.2) 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 to obtain relation-maintenance formulas for the newly introduced instrumentation relations, as well as for those instrumentation relations whose definitions have been changed [25].
- (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 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.

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, because of the comparison in line [10] of Fig. 7, 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 $inOrder_{dle,n}(u)$.

An algorithm to generate new instrumentation relations should take into account the sources of imprecision. §4.1 describes subformula-based refinement; in this method, query subformulas that are responsible for an indefinite answer are used to generate new instrumentation relations. §5 discusses a shortcoming of subformula-based refinement and describes ILP-based refinement.

4.1 Subformula-Based Refinement

The subformulas of the query that are responsible for the indefinite answer are good candidates for defining new instrumentation relations. Fig. 11 presents function *instrum*, a recursive-descent procedure to generate defining formulas for new instrumentation

φ	return value of $instrum(\varphi, S, Z)$
0, 1	ERROR
1/2	\emptyset
$v_1 = v_2$	\emptyset
$p(v_1, \dots, v_k)$	$(p \in \mathcal{C}) ? \emptyset : instrum(\psi_p, S, Z)$ if $(k = 1 \wedge p \notin \mathcal{A}) \mathcal{A} := \mathcal{A} \cup \{p\}$
$\neg \varphi_1$	$instrum(\varphi_1, S, Z)$
$\varphi_1 \vee \varphi_2$	$(\varphi \in \{\psi_p \mid p \in \mathcal{I}\}) ? \emptyset : \{\varphi\}$ $\cup ((\llbracket \varphi_1 \rrbracket_3^S(Z) = 1/2) ? instrum(\varphi_1, S, Z) : \emptyset)$
$\varphi_1 \wedge \varphi_2$	$\cup ((\llbracket \varphi_2 \rrbracket_3^S(Z) = 1/2) ? instrum(\varphi_2, S, Z) : \emptyset)$
$\exists v : \varphi_1$	$(\varphi \in \{\psi_p \mid p \in \mathcal{I}\}) ? \emptyset : \{\varphi\}$ $\cup \cup_{u \in S} ((\llbracket \varphi_1 \rrbracket_3^S(Z[v \mapsto u]) = 1/2$
$\forall v : \varphi_1$	$? instrum(\varphi_1, S, Z[v \mapsto u])$ $: \emptyset)$
$p^*(v_1, \dots, v_k)$	$(\varphi \in \{\psi_q \mid q \in \mathcal{I}\}) ? \emptyset : \{\varphi\}$ $\cup \cup_{\substack{u'_1, u'_2 \in S, \\ u'_1 \neq u'_2}} ((\llbracket p \rrbracket_3^S(Z[u'_1 \mapsto u'_1, u'_2 \mapsto u'_2]) = 1/2)$ $? instrum(p, S, Z[u'_1 \mapsto u'_1, u'_2 \mapsto u'_2])$ $: \emptyset)$

Figure 11: Function $instrum$, which looks for formulas to be used as definitions of new instrumentation relations.

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 for which $\llbracket \varphi \rrbracket_3^S(Z) = 1/2$.

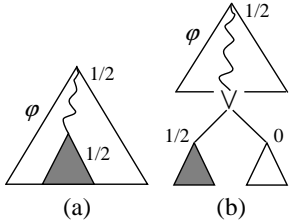


Figure 10: (a) Recursive-descent function $instrum$ finds the subformulas of φ that can cause the 1/2 answer. (b) Ex.: imprecision in an or-subformula.

and is not in the set of abstraction relations, add it to the set of abstraction relations.

$instrum(p \in \mathcal{I}, \dots)$ Examine ψ_p , the defining formula of p . Also, if p is unary and is not in the set of abstraction relations, add it to the set of abstraction relations.

$instrum(\varphi_1 \vee \varphi_2, \dots)$ If φ (i.e., $\varphi_1 \vee \varphi_2$) does not define an instrumentation relation, it will be used as the definition of a new instrumentation relation. Also, examine 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 (i.e., $\exists v : \varphi_1$) does not define an instrumentation relation, it will be used as the definition of a new instrumentation relation. Also, examine 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 $q(v_1, \dots, v_k)$, where v_1, \dots, 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 the second line of entry $p(v_1, \dots, v_k)$ in Fig. 11, which handles

A precondition of $instrum$ is that $\llbracket \varphi \rrbracket_3^S(Z) = 1/2$. Starting with this assumption, $instrum$ attempts to find subformulas of φ that, if sharpened, would sharpen the value of the whole formula (see Figs. 10 (a) and (b)). If such subformulas are found, they will be used to define new instrumentation relations. Below are explanations of a few cases:

$instrum(1, \dots)$ This violates the precondition of $instrum$.

$instrum(1/2, \dots)$ Nothing can be done in this case.

$instrum(p \in \mathcal{C}, \dots)$ If p is unary

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 $instrum$.

core and instrumentation relations).

Example. As we saw in §3, abstract interpretation collects 3-valued structure S_4 of Fig. 4 at the exit node of `InsertSort`. 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 are 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 [15]. Note also that $instrum$ returns no subformulas of the definition of $r_{n,x}$. This is because $r_{n,x}(v)$ evaluates to a definite value (1) for both $v \mapsto u_{23}$ and $v \mapsto u_1$ (see Fig. 4).

4.2 Refinement of the Program's Actions

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 the instrumentation relations introduced by abstraction 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 this end, for each new

relation $p(v_1, \dots, v_k)$, the query and all other instrumentation relations' defining formulas are scanned for occurrences of ψ_p . Every occurrence of $\psi_p\{w_1/v_1, \dots, w_k/v_k\}$, i.e., ψ_p with w_i substituted for free variable v_i , is replaced with $p(w_1, \dots, w_k)$, thus enabling the use of stored value $p(w_1, \dots, w_k)$ in place of the evaluation of ψ_p .

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 [25].

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 $\psi_{sorted_1}, \dots, \psi_{sorted_4}$ (in that order). 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 Tab. 4.

4.3 Refinement of the Abstract Input

Before performing abstract interpretation of the refined transition system, 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 $p(v_1, \dots, v_k)$, 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, \dots, v_k results in safe values, these

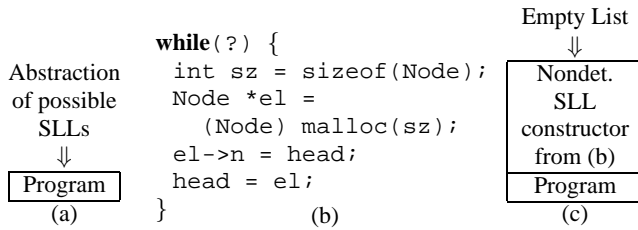


Figure 12: Illustration of input specifications for programs that manipulate singly-linked lists. (a) Traditional input specification in TVLA. (b) A fragment of code that nondeterministically constructs all possible singly-linked lists. (c) The use of loop (b) to specify the set of inputs.

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.⁴

$$\forall v_1, v_2 : (dle(v_1, v_2) \wedge dle(v_2, v_1) \wedge t_n^0(v_1, v_2)) \rightarrow t_n(v_1, v_2) \quad (5)$$

The first run of abstract interpretation on `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, $stable_2(v_1, v_2)$:

$$(dle(v_1, v_2) \wedge dle(v_2, v_1) \wedge t_n^0(v_1, v_2)) \rightarrow t_n(v_1, v_2) \quad (6)$$

Consider the rightmost structure of Fig. 8,⁵ 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 $dle(u_s, u_s)$, $t_n^0(u_s, u_s)$, 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 **Data-Structure Constructor**, 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. 12 illustrates the idea.

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 $r_{n,x}^0(v)$ 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

⁴A related property, antistability, asserts that the order of elements with equal data-components is reversed:

$$\forall v_1, v_2 : (dle(v_1, v_2) \wedge dle(v_2, v_1) \wedge t_n^0(v_1, v_2)) \rightarrow t_n(v_2, v_1)$$

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. 7 (i.e., when looking for the correct place to insert the current node). This implementation of insertion sort is antistable.

⁵In that structure, all history relations, such as t_n^0 , have the same values as their active counterparts, but have been omitted from the figure for clarity.

is about to be performed on the procedure proper.

The $stable_2$ instrumentation relation defined by Formula (6) exemplifies the benefits of the DSC methodology. The maintenance of $stable_2$, t_n , t_n^0 , 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 run of abstract interpretation (line [1] in Fig. 9). 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.

4.4 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 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.

5. ILP-BASED REFINEMENT

5.1 Shortcomings of the Strategy of §4.1

Procedure `InsertSort` consists of two nested loops (see Fig. 7). 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.⁶

5.2 Learning Instrumentation Relations

Fig. 13 shows the structure S_{13} , which arises during abstract interpretation just before line [6] of Fig. 7, 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_{14} , shown in Fig. 14. Bold entries of tables in Fig. 13 indicate definite values that are transformed into 1/2 in S_{14} . Structure S_{13} satisfies the sortedness invariant discussed above: every node among u_1, \dots, u_4 has the dle relationship with all nodes

⁶In contrast, subformula-based refinement is capable of establishing the antistability of `InsertSort_AS`. When looking for a place to insert r ’s target, this routine stops when $l->data \geq r->data$ 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 refinement introduces the appropriate instrumentation relations based on subformulas of the antistability query, TVLA is able to establish antistability of `InsertSort_AS`.

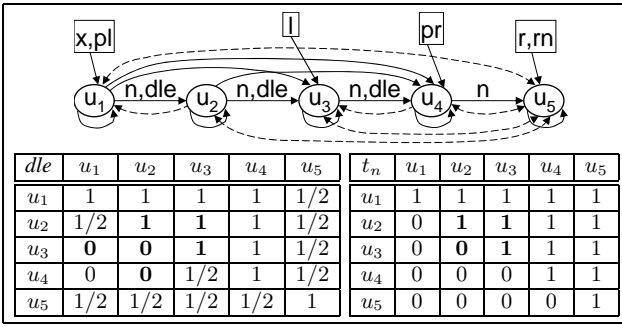


Figure 13: Structure S_{13} , which arises just before line [6] of Fig. 7. (All unlabeled edges between nodes represent the dle relation.)

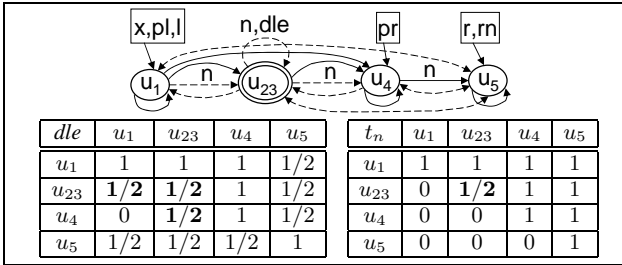


Figure 14: Structure S_{14} , corresponding to the transformation of S_{13} by the statement on line [6] of Fig. 7. (All unlabeled edges between nodes represent the dle relation.)

appearing later in the list, except r 's target, u_5 . However, a piece of this information is lost in structure S_{14} : $dle(u_{23}, u_{23}) = 1/2$, 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 [27]. 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 .

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 . A modified version of the algorithm described in §2.2 can be used to learn formulas for the following three kinds of relations:

Type I: Unary relation r_1 with positive example $\{u\}$ for one $u \in U_1$, and negative examples $U_1 - \{u\}$.

Type II: Unary relation r_2 with positive examples U_1 .

Type III: Binary relation r_3 with positive examples $U_1 \times U_1$.

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 .⁷

For the background logical structure that serves as input to ILP,

⁷Type III relations can be generalized to ternary and higher-arity relations.

we pass the structure S_1 identified at an information-loss point. We restrict the algorithm to use only active 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 (1 or 0) 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 in two ways. First, we 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. 6), 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.

The second change is needed to handle the lack of negative examples for relations of Types II and III. We change the inner loop (line [5] of Fig. 6) from a while loop to a do-while, so that we obtain non-trivial formulas even in the absence of negative examples. We also replace FOIL's information-gain heuristic with a simpler heuristic based on the percentage of positive examples covered by the new disjunct.

We now describe how this variant of ILP is able to learn a useful binary formula using structure S_{13} of Fig. 13. The set of individuals of S_{13} 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 $dle(v_1, v_2)$ 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, $dle(v_1, v_2)$ 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

$$\psi_{r_3}(v_1, v_2) \stackrel{\text{def}}{=} dle(v_1, v_2) \vee \neg t_n(v_1, v_2), \quad (7)$$

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

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 $dle(u_{23}, u_{23})$ is $1/2$ in S_{14} , $r_3(u_{23}, u_{23})$ 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 $dle(v_1, v_2) \vee t_n(v_2, v_1)$ are also learned using ILP (cf. Fig. 17). Not all of them are useful to the verification process, but introducing unnecessary instrumentation relations cannot harm the analysis, aside from increasing its cost.

5.3 ILP and the Refinement Loop

ILP gives us a powerful mechanism for learning new abstractions. At present, we employ subformula-based refinement first, because the cost of this strategy is reasonable (see §6) and the strategy is often successful. If during refinement the call to *instrum* on line [6] of Fig. 9 returns no formulas and adds no relations to the set of abstraction relations, we turn to the ILP strategy.

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 §5.2. Our present implementation only attempts to learn binary formulas (i.e., of Type III). We prune the returned set of formulas to lower the cost of the analysis. We first remove formulas defined in terms of a single relation symbol. Such formulas are usually tautologies (e.g., $dle(v_1, v_2) \vee dle(v_2, v_1)$). We then define new instrumentation relations using only learned formulas of a simple

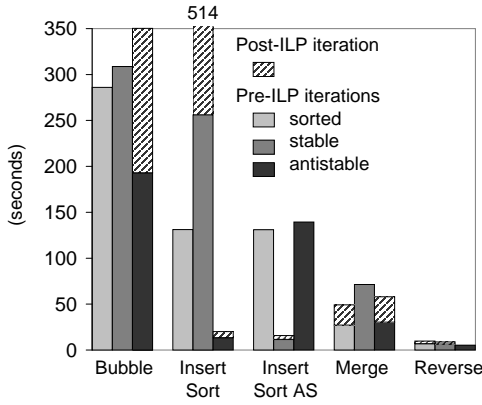


Figure 16: Execution times. For each program, the three bars represent the sorted, stable, and antistable queries. In cases where subformula-based refinement failed, the upper portion of the bars shows the cost of the last iteration of the analysis (on both the DSC and the program) together with the ILP cost.

form (currently, those with two atomic subformulas). We finally use these relations to refine the abstraction by performing the steps of lines [7] and [8] of Fig. 9, as done for subformula-based refinement.

When attempting to verify the stability of `InsertSort`, nine formulas are learned using the ILP algorithm, among them Formula (7). Upon completion of the refinement steps, the subsequent run of the analysis successfully verifies the stability of `InsertSort`.

6. EXPERIMENTAL EVALUATION

To evaluate the method presented in this paper, we extended TVLA to perform iterative abstraction refinement, and applied it to three queries and five programs (see Fig. 15). Besides `InsertSort`, the test programs included sorting procedures `BubbleSort` and `InsertSort_AS`, list-merging procedure `Merge`, and *in-situ* list-reversal procedure `Reverse`.

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

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

Figure 15: Results from applying iterative abstraction refinement to the verification of properties of programs that manipulate linked lists.

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. 15 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.

Fig. 17 shows the numbers of instrumentation relations used during the last iteration of abstraction refinement. The number of ILP-

Fig. 15 shows that our 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

Test Program	sorted	stable	antistable
	# instrum rels total/ILP	# instrum rels total/ILP	# 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

Figure 17: The numbers of instrumentation relations (total and learned by ILP) used during the last iteration of abstraction refinement.

learned relations used by the analysis is small relative to the total number of instrumentation relations.

Fig. 16 gives execution times that were collected on a 3GHz Linux PC with 3.75GB of RAM. 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].⁸ The maximum amount of memory used by TVLA to perform the analyses varied from just under 2 megabytes to 32 megabytes.⁹

The cost of the invocations of the ILP algorithm when attempting to verify the antistability of `BubbleSort` was 25 seconds. This cost was incurred at 133 information-loss points. For all other benchmarks, the ILP cost was less than ten seconds. We consider these costs to be low given that our implementation of the ILP algorithm is unoptimized and we take no advantage of the fact that an ILP computation at one information-loss point often produces the same results as ILP computations at other information-loss points.

We performed three additional experiments to test 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 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, subformula-based refinement successfully verified the query that the nodes of the tree pointed to by variable τ remain in sorted order at the end of the programs:

$$\forall v_1 : r_\tau(v_1) \rightarrow (\forall v_2 : (left(v_1, v_2) \rightarrow dle(v_2, v_1)) \wedge (right(v_1, v_2) \rightarrow dle(v_1, v_2))). \quad (8)$$

The initial specifications for the analyses included only three standard instrumentation relations, similar to those listed in Tab. 2. Relation $r_\tau(v_1)$ from Formula (8), for example, distinguishes nodes in the (sub)tree pointed to by τ . The DSC used for the analyses non-deterministically constructs a binary-search tree by allocating one new node at a time and inserting it into the tree in the appropriate position according to its `data`-value. 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.

⁸Sortedness is the only query in our set to which TVLA has been applied before this work.

⁹TVLA is written in Java. Here we report the maximum of total memory minus free memory, as returned by Runtime.

7. RELATED WORK

The work reported here is similar in spirit to counterexample-guided abstraction refinement [12, 5, 13, 21, 7, 3, 9]. A key difference between our setting and that explored in prior work is the abstract domain. All abstraction-refinement work to date in the model-checking community 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 a refinement method 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 in a refinement algorithm that addresses the richer class of abstractions that TVLA supports.

A second distinguishing feature of our work is that the method is not driven by counterexample traces but rather 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). The SLAM toolkit identifies the shortest prefix of a spurious counterexample trace that cannot be extended to a feasible path; in general, the first information-loss point occurs before the end of the prefix. Information-loss-guided refinement, on the other hand, can identify the earliest points where abstraction loses information. It can be used to find new instrumentation relations, as well as the earliest points at which they should become part of the abstraction. The latter information is likely to be important when we try to extend our work to learn the first-order analog of Blast’s parsimonious abstractions [9].

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. However, the abstract domain that we work with has 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”) [8]. 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.

In [24], weakest preconditions are used to generate nullary instrumentation relations, which are then generalized manually. The 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 the binary relation $r_3(v_1, v_2)$, defined by Formula (7), which allows the analysis to establish the stability of `InsertSort`.

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 [28].

Other work that relates machine-learning techniques and program analysis includes [2, 17, 26]. The Strauss tool [2] uses a machine-learning approach to discovering specifications of API protocols. The underlying premise is that even programs with bugs contain hints that can reveal correct protocols. The Cooperative Bug Isolation project [17] instruments programs and collects information about their executions. Statistical and machine-learning

techniques are used to find bugs by mining the information about crashing and non-crashing runs. The technique for finding the most-precise abstract value for a set of concrete stores (expressed as a logical formula) successively approximates the result from below [26]; this technique is related to algorithm Find-S from machine learning [18, §2.4] —they both search a space of hypotheses to find the most specific hypothesis that satisfies the positive examples (the input concrete stores).

Our work represents a new connection between program analysis and machine learning: it shows how ILP can be used as part of an abstraction-refinement loop to learn an appropriate abstraction.

8. REFERENCES

- [1] TVLA system. <http://www.cs.tau.ac.il/tvla/>.
- [2] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.
- [3] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, pages 103–122, 2001.
- [4] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, 1990.
- [5] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.
- [6] C. Colby, P. Godefroid, and L. Jagadeesan. Automatically closing open reactive programs. In *PLDI*, pages 345–357, 1998.
- [7] S. Das and D. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD*, pages 19–32, 2002.
- [8] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.
- [9] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
- [10] B. Jeannot, N. Halbwegs, 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*, chapter 4, 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*, 2000.
- [16] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.
- [17] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.
- [18] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [19] S. Muggleton. Inductive logic programming. *New Generation Comp.*, 8(4):295–317, 1991.
- [20] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *J. Logic Prog.*, 19/20:629–679, 1994.
- [21] C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible counter-examples when model checking Java programs. In *TACAS*, pages 284–298, 2001.
- [22] J.R. Quinlan. Learning logical definitions from relations. *Mach. Learn.*, 5:239–266, 1990.
- [23] J.R. Quinlan and R.M. Cameron-Jones. Induction of logic programs: FOIL and related systems. *New Gen. Comp., Special issue on ILP*, 13(3-4):287–312, 1995.
- [24] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *PLDI*, pages 83–94, 2002.
- [25] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas with applications to program analysis. In *ESOP*, 2003.
- [26] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.
- [27] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
- [28] O. Tkachuk, M. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *ASE*, 2003.