Putting Static Analysis to Work for Verification: A Case Study

Tal Lev-Ami* Thomas Reps† Mooly Sagiv‡,* Reinhard Wilhelm§

Abstract

We study how program analysis can be used to:

- Automatically prove partial correctness of correct programs.
- Discover, locate, and diagnose bugs in incorrect programs.

Specifically, we present an algorithm that analyzes sorting programs that manipulate linked lists. A prototype of the algorithm has been implemented.

We show that the algorithm is sufficiently precise to discover that (correct versions) of bubble-sort and insertion-sort procedures do, in fact, produce correctly sorted lists as outputs, and that the invariant "is-sorted" is maintained by list-manipulation operations such as element-insertion, element-deletion, and even destructive list reversal and merging of two sorted lists. When we run the algorithm on erroneous versions of bubble-sort and insertion-sort procedures, it is able to discover and sometimes even locate and diagnose the error.

1 Introduction

This paper shows that static analysis can be employed to

- Automatically prove partial correctness of correct programs.
- Discover, locate, and diagnose bugs in incorrect programs.

While static analysis has been used previously to find potential bugs ("program anomalies") as well as to demonstrate the absence of bugs in programs that manipulate scalar variables and arrays [12, 1], the present paper concerns programs that manipulate pointers and heap-allocated storage. This paper demonstrates that it is possible to create analysis algorithms that are of sufficient precision that the program's partial correctness can be established from the information contained in the "state-descriptors" obtained via static analysis.

The approach to verification described here uses a method for creating program-analysis algorithms that was described in [25, 26] (see Section 2). To illustrate how the verification method works, we use an extended version of the "shape analysis" described in [25, 26] (which determines information about the shapes of the heap-allocated data structures that a pointer variable can point to). In the present paper, shape descriptors are extended with information that keeps track of the relative order of the values in data fields of neighboring list elements. By carrying out static analysis with this family of shape descriptors, we are able to verify the correctness of several versions of a sorting program that operates on linked lists.

The most important characteristics of our method stand out if we contrast it with conventional approaches to program verification. Ordinarily, program verification involves establishing that a given program satisfies a user-supplied *specification*. With the standard approach to proving correctness, the user supplies a pre-condition and post-condition for each procedure, as well as a loop-invariant for each loop in the program [9, 13]. Taken together, these break the program into a collection of finite-length path fragments whose correctness must be established to prove the overall correctness of the program. The verification system traverses the program to gather up a collection of *verification conditions*—one for each path fragment. The system then calls a theorem prover to establish that each verification condition is a theorem [17, 8].

In this paper, we make use of quite different machinery in order to establish that a program works correctly:

- The user supplies a "descriptor" of the acceptable inputs to the program.
- An abstract interpretation of the program is performed.
- The descriptor associated with the program's exit point is checked to make sure that only acceptable outputs can be produced.

Thus, with our method, the specification takes the form of an input descriptor and an acceptability criterion on output descriptors, but loop invariants are completely omitted.

The latter feature, in particular, is a highly desirable characteristic: Whereas a precise statement of a desired input-output relationship is something that any verification method will require, the experience of the last thirty years is that loop invariants impose such a burden on the programmer that any method that requires them has, at best, a limited market for adoption.

The above characterization of our work no doubt raises two questions in the reader's mind:

- How much is being swept under the rug by the phrase "An abstract interpretation of the program is performed"?
- How can a verification method possibly avoid using loop invariants?

The answers to these questions are as follows:

- *Abstract interpretation*: It is important that the *right* abstract interpretation be performed—in particular, one that applies to descriptors that are sufficiently expressive to maintain the distinctions needed to allow the whole enterprise to succeed. The specification of such

*Dept. of Comp. Sci.; Tel-Aviv Univ.; Tel-Aviv 69978, Israel; {tla,sagiv}@math.tau.ac.il

†Comp. Sci. Dept.; Univ. of Wisconsin; Madison, WI 53706; USA; reps@cs.wisc.edu. Supported in part by the NSF under grant CCR-9619219 and by the U.S.-Israel BSF under grant 96-00337.

‡Supported in part by the U.S.-Israel BSF under grant 96-00337.

§Informatik; Univ. des Saarlandes; 66123 Saarbrücken; Germany; wilhelm@cs.uni-sb.de

an analysis becomes one of the skills needed to apply our verification methodology.[1]

However, this seems to present a significant new obstacle to adoption: With the methodology of abstract interpretation, it is typically not an easy task to obtain appropriate abstract state-transformation functions and show that they are correct. On the contrary, papers on program analysis often contain exhaustive (and exhausting) proofs to demonstrate that a given abstract semantics provides answers that are safe with respect to a given concrete semantics.

What allows us to overcome this obstacle is a method for creating program-analysis algorithms set forth in a previous paper of ours [25, 26]. With this technique, the developer of a program-analysis algorithm is freed from most of the proof obligations normally associated with abstract interpretation. A further advantage of this approach is that it provides a *parametric* framework for program analysis. That is, it provides a method for generating a program-analysis tool from a high-level, user-supplied description of what is desired.

- *Loop invariants*: In the conventional approach to program verification, loop invariants serve to break the program into a finite number of finite-length path fragments. In contrast, our approach uses *abstraction* to allow the entire program to be "executed" on (finite representations of) the infinity of possible runtime stores. In some sense, our approach performs a kind of "state-space exploration", and hence is related to model checking [5]. By limiting the abstract execution of the program to an *a priori* finite set of store descriptors, convergence to a fixed-point is guaranteed.

  The descriptors that arise at the head of each loop can be considered to be loop invariants—but ones that have been automatically *inferred* by our system, not provided by the user. Because our domain of descriptors is finite, this invariant-synthesis problem has a different character than those used in [4, 29, 30, 28] (i.e., in our work, the invariants are generated in the course of performing an iterative fixed-point computation). Furthermore, there is nothing special about the program points at the heads of loops; they are treated in exactly the same way that any other program point is treated during the analysis.

At a technical level, the approach taken in this paper is much different from that used in conventional approaches to program verification, where assertions (formulae) are pushed backwards through statements. The justification for propagating information in the backwards direction is that it avoids the existential quantifiers that arise when assertions are pushed in the forwards direction to generate strongest postconditions. Ordinarily, strongest postconditions present difficulties because quantifiers accumulate, forcing one to work with larger and larger formulae. The abstract-interpretation method from [25, 26] pushes information in the forwards direction. Furthermore, as discussed in Sections 2 and 6, it works at the semantic level; that is, it operates directly on explicit representations of logical structures, rather than on implicit representations, such as logical formulae. Because the analysis is carried out with respect to a domain of state descriptors that are *a priori* of bounded size, forwards propagation cannot generate shape descriptors of unbounded size, and the analysis is guaranteed to terminate.

[1] The abstract interpretation is actually designed not for a specific *program*, but for a particular *datatype*—once designed, it can be used in proving the correctness of multiple programs that manipulate data of that type.

```
/* list.h */
typedef struct node {
  int d;
  struct node *n;
} *L;
```

Figure 1: A type declaration for singly linked lists.

The notion of an *instrumentation predicate* plays a key role in our work. An instrumentation predicate captures a property that an *individual storage element* may or may not possess. In general, adding additional instrumentation predicates refines the abstraction used for program analysis; it yields a more precise analysis algorithm that maintains finer distinctions, and hence allows more questions about the program's data structures to be answered.

From the perspective of someone interested in verifying a program via our approach, the need to adopt a local, element-wise view of a data structure gives the approach a markedly different flavor than conventional approaches to program verification, where the emphasis is on developing *invariants*. For instance, the notion of an instrumentation predicate can be contrasted with that of a *datatype invariant* (e.g., see [15]):

- A datatype invariant states a *global* property of an abstract datatype's instances that holds on entry to and exit from the datatype's operations.
- An instrumentation predicate captures a *local* property that can be used to distinguish among some of a datatype's components.

As will be discussed in Sections 3.1 and 6, with our approach it is also necessary to specify how the local properties of interest are affected by the execution of each kind of statement in the programming language.

We illustrate the use of program analysis for verification by means of an extended example—the analysis of several versions of a sorting program that operates on linked lists. We show that the verification method is sufficiently precise to discover that (correct versions) of bubble-sort and insertion-sort procedures do, in fact, produce correctly sorted lists as outputs, and that the invariant "is-sorted" is maintained by list-manipulation operations, such as element-insertion, element-deletion, destructive list reversal, and merging of two sorted lists. When we run the algorithm on erroneous versions of bubble-sort and insertion-sort procedures, it is able to discover and sometimes even locate and diagnose the error.

Figure 1 shows a declaration of a linked-list type; Figure 2 shows an implementation of an insertion-sort algorithm; Figure 3 contains the main program analyzed by our algorithm. The C code for procedures `create`, `merge`, and `reverse` is given in Appendix A.

The remainder of the paper is organized into five sections: Section 2 summarizes the program-analysis framework of [25, 26], which shows how 3-valued logic can serve as a basis for program analysis. Section 3 describes how this approach can be used to show that a sorting procedure is *partially correct*, i.e., if the procedure terminates, then the resulting list is sorted in increasing order. Section 4 discusses the behavior of the analysis algorithm on incorrect procedures. Section 5 reports on an implementation of the method using the TVLA sys-

```
/* insertion.c */
#include "list.h"
L insert_sort(L x) {
  L r, pr, rn, l, pl;
  r = x;
  pr = NULL;
  while (r != NULL) {
     l = x;
     rn = r ->n;
     pl = NULL;
     while (l != r) {
        if (l->data > r->data) {
          pr->n = rn;
          r->n = l;
          if (pl == NULL)
            x = r;
          else
            pl->n = r;
          r = pr;
          break;
        }
        pl = l;
        l = l->n;
     }
     pr = r;
     r = rn;
  }
  return x;
}
```

Figure 2: A correct version of insertion sort.

```
/* main.c */
#include "list.h"
int main() {
  L x, y, z, w;
  L create(), insert_sort(L);
  L merge(L,L), reverse(L);

  x = create(); l1:
  x = insert_sort(x); l2:

  y = create(); l3:
  y = insert_sort(y); l4:

  z = merge(x,y); l5:
  w = reverse(z); l6:
}
```

Figure 3: A program that performs several operations on sorted lists.

tem [18, 19]. Section 6 discusses limitations of our approach, related work, and future directions.

2 The Use of 3-Valued Logic for Program Analysis

In this section, we summarize the framework presented in [25, 26], where we showed how 3-valued logic can serve as the basis for program analysis. A generalized version of that analysis framework has been implemented in a system called TVLA [18, 19] (for **T**hree-**V**alued-**L**ogic **A**nalyzer). TVLA was used to implement the verification method described in this paper (and to generate all of the figures presented). Where relevant, features specific to TVLA will be noted below.

Kleene's 3-valued logic is an extension of ordinary 2-valued logic with the special value of 1/2 (unknown) for cases in which predicates could have either value, i.e., 1 (true) or 0 (false). Kleene's interpretation of the propositional operators is given in Table 3. We say that the values 0 and 1 are *definite values* and that 1/2 is an *indefinite value.*

2.1 Representing Memory States via Logical Structures

A 2-*valued logical structure* S is comprised of a set of individuals (nodes) called a universe, denoted by U^S, and an interpretation over that universe for a set of predicate symbols. The interpretation of a predicate symbol p in S is denoted by p^S. For every predicate p of arity k, p^S is a function $p^S\colon (U^S)^k \rightarrow \{0,1\}$. 2-valued structures are used to represent memory states used in the operational semantics of the program.

2-valued logical structures will be depicted as directed graphs in this paper.[2] A directed edge between nodes u_1 and u_2 that is labeled with binary predicate symbol p indicates that $p^S(u_1, u_2) = 1$. Also, for a unary predicate symbol p, we write p inside a node u when $p^S(u) = 1$; conversely, we write $p = 0$ inside node u when $p^S(u) = 0$.

The set of predicate symbols is partitioned into two disjoint sets: *core* and *instrumentation* predicate symbols. Core predicates are part of any pointer semantics. They record atomic properties of the memory state. Instrumentation predicates are used to record derived properties. They have a defining formula in terms of the core predicates. Evaluating the formula for an instrumentation predicate i in a structure S yields its value i^S. The operational semantics of a statement is specified by *predicate-update formulae*: These say how the values of the predicates change when the statement is executed.

In this paper, a 2-valued structure represents a memory state (also called a *store*); an individual corresponds to a list element. The intended meaning of the core predicates is given in Table 1, and the intended meaning of the instrumentation predicates is given in Table 2 (for the moment ignore the third column). The store in Figure 4 is represented by the 2-valued structure S_5 shown in Figure 5. The structure S_5 has four nodes, u_0, u_1, u_2, and u_3, which represent the four list elements. This representation intentionally ignores the specific values of the `d`- and the `n`-components (an `int` and a memory address, respectively), and just records certain relationships that hold among list elements:

- The binary relation n captures whether one list element is the successor of another.
- The binary relation *dle* keeps track of the relative order between two list elements' `d`-fields.

For each pointer variable `x`, there is a unary predicate x. The value of $x^S(u)$ is 1 if variable `x` points to the list element represented by u. In Figure 5, the unary predicate is 1 only for u_0. To make the figures more intuitive, the value of the x-predicate is depicted via an edge from a box labeled x to the node that `x` points to (and via the absence of edges from

[2] We only use predicates of arity ≤ 2.

Predicate	**Intended Meaning**	**Defining Formula**
$r[n,x](v)$	Is v reachable from program variable `x` using component `n`?	$\exists v_1 : (x(v_1) \wedge n^*(v_1, v))$
$c[n](v)$	Does v reside on a directed cycle of `n`-components?	$n^+(v, v)$
$is[n](v)$	Is v pointed to by more than one `n`-component?	$\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$
$inOrder[dle, n](v)$	Is v the first of a pair of neighbors with non-decreasing `d`-fields?	$\forall v_1 : n(v, v_1) \Rightarrow dle(v, v_1)$
$inROrder[dle, n](v)$	Is v the first of a pair of neighbors with non-increasing `d`-fields?	$\forall v_1 : n(v, v_1) \Rightarrow dle(v_1, v)$

Table 2: The instrumentation predicates used in this paper and their meaning. There is a separate predicate $r[n, x]$ for every program variable `x`. The defining formulae are explained in Section 2.3.

Predicate	**Intended Meaning**
$x(v)$	Is v pointed to by variable `x`?
$n(v_1, v_2)$	Does the `n`-component of v_1 point to v_2?
$dle(v_1, v_2)$	Is the `d`-component of v_1 less-than-or-equal-to the `d`-component of v_2?

Table 1: The core predicates used in the analysis. There is a separate predicate x for every program variable `x`.

$\wedge$	0	1	1/2
0	0	0	0
1	0	1	1/2
1/2	0	1/2	1/2

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

$\neg$	
0	1
1	0
1/2	1/2

Table 3: Kleene's 3-valued interpretation of the propositional operators.

the box for x to the nodes that `x` does not point to), rather than by placing the value of $x^S(u)$ inside each node u.

Pointer components within the list elements are represented as binary predicates (i.e., $n^S(u_1,\ u_2) = 1$ if the `n`-component of u_1 points to u_2). Also, inequalities between list elements are represented by the binary predicate dle (i.e., $dle^S(u_1,\ u_2) = 1$ if the `d`-component of u_1 is less than or equal to the `d`-component of u_2).

The unary instrumentation predicate $r[n, x](v)$ holds for list elements that are reachable from program variable `x`, possibly using a sequence of accesses through `n`-components. In structure S_5 in Figure 5, $r[n, x]^{S_5}$ is 1 for all of the nodes because they are all reachable from `x`.

An important aspect of explicitly storing $r[n, x]$ and other instrumentation predicates is that we can compute the effect of a program statement on the predicates' values without reevaluating the instrumentation predicates' defining formulae. For instance, for the statement `y = x`, the nodes reachable from `y` after the statement executes are the same as the nodes reachable from `x`. There is no need to reevaluate the formula defining what it means to be reachable from `y`; instead, we can generate $r'[n, y]$, the value of predicate $r[n, y]$ in the state after the statement executes, via the predicate-update formula $r'[n, y](v) = r[n, x](v)$.

The instrumentation predicate $is[n]$ holds for nodes shared by `n`-components. (A node is *shared* by `n`-components if it is pointed to by more than one list elements' `n`-component.) In Figure 5, all the elements of the list are unshared, and thus $is[n]^{S_5}$ is 0 for all of them. In fact, throughout the execution of all of the example programs in the paper, $is[n]^S$ is always 0, for all nodes.

The instrumentation predicate $c[n]$ holds for nodes on a cycle of `n`-components. The cyclicity instrumentation is used to avoid performing a transitive-closure operation when updating the reachability information [25]. In Figure 5, the list

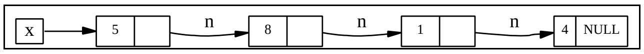


Figure 4: A possible store for a linked list.

is acyclic, and thus $c[n]^{S_5}$ is 0 for all of the nodes.

To express sortedness of lists we use the instrumentation predicates $inOrder[dle, n]$ and $inROrder[dle, n]$. Predicate $inOrder[dle, n]$ holds for nodes whose `d`-components are less than or equal to those of their `n`-successor. Similarly, $inROrder[dle, n]$ holds for nodes whose `d`-components are greater than or equal to those of their `n`-predecessors.

TVLA makes an explicit assumption that the set of predicate symbols used throughout the analysis is fixed. (The number of individuals in structures can vary throughout the analysis.)

2.2 Conservative Representation of Sets of Memory States via 3-Valued Structures

Like 2-valued structures, a 3-*valued logical structure* S is also comprised of a universe U^S, and an interpretation of the predicate symbols. However, for every predicate p of arity k, p^S is a function $p^S\colon (U^S)^k \to \{0, 1, 1/2\}$, where 1/2 explicitly captures unknown predicate values.

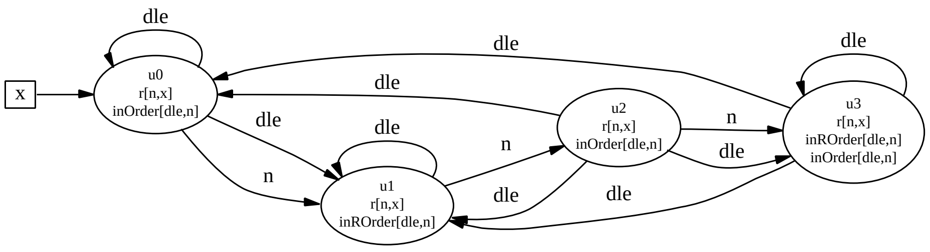
3-valued logical structures are also drawn as directed graphs. Definite values are drawn as in the 2-valued structures. Binary indefinite (1/2) predicate values are drawn as dotted directed edges. Also, we write $p = 1/2$ inside a node u when $p^S(u) = 1/2$.

Let $S^\natural$ be a 2-valued structure, S be 3-valued structure, and $f\colon U^{S^\natural} \to U^S$ be a surjective function. We say that f *embeds* $S^\natural$ *into* S if for every predicate p of arity k and $u_1, u_2, \ldots, u_k \in U^{S^\natural}$, either $p^{S^\natural}(u_1, u_2, \ldots, u_k) = p^S(f(u_1), f(u_2), \ldots, f(u_k))$ or $p^S(f(u_1), f(u_2), \ldots, f(u_k)) = 1/2$. We say that S *conservatively represents all the* 2*-valued structures that can be embedded into it by some function* f. Thus, S can compactly represent many structures.

Example 2.1 The 3-valued structure S_6 shown in Figure 6 represents the 2-valued structure S_5 for $f(u_0) = u_0$ and $f(u_1) = f(u_2) = f(u_3) = u$. In fact, the structure shown in Figure 6 represents all lists with two or more elements.

The unary predicate symbol x has $x^{S_6}(u_0) = 1$, indicating that the program variable `x` is known to point to the list element represented by $u0$, and $x^{S_6}(u) = 0$, indicating that `x` is known not to point to any of the list elements represented by u.

The unary predicates $inOrder[dle, n]$ and $inROrder[dle, n]$ both have value 1/2, for both u_0 and u, which indicates that nothing is known about the relative order of the values of the `d`-fields of neighboring elements in the list.

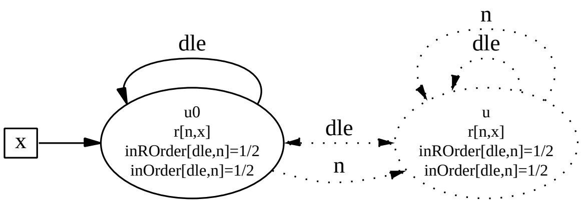


	x	$r[n,x]$	$is[n]$	$c[n]$	$inOrder[dle,n]$	$inROrder[dle,n]$
$u0$	1	1	0	0	1	0
$u1$	0	1	0	0	0	1
$u2$	0	1	0	0	1	0
$u3$	0	1	0	0	1	1

n	$u0$	$u1$	$u2$	$u3$
$u0$	0	1	0	0
$u1$	0	0	1	0
$u2$	0	0	0	1
$u3$	0	0	0	0

dle	$u0$	$u1$	$u2$	$u3$
$u0$	1	1	0	0
$u1$	0	1	0	0
$u2$	1	1	1	1
$u3$	1	1	0	1

Figure 5: A logical structure S_5 representing the store shown in Figure 4 in graphical and tabular representations.



	x	$r[n,x]$	$is[n]$	$c[n]$	$inOrder[dle,n]$	$inROrder[dle,n]$
$u0$	1	1	0	0	1/2	1/2
u	0	1	0	0	1/2	1/2

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

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

Figure 6: A 3-valued structure S_6 representing arbitrary lists of length 2 or more that are pointed to by program variable `x`.

The `n`-edges from $u0$ to u and from u to u are dotted, indicating that two of the entries for binary predicate symbol n have indefinite values: $n^{S_6}(u_0, u) = 1/2$, indicating that the list element represented by u_0 may point to a list element represented by u—namely, the second list element (u_1 in Figure 5). Also, $n^{S_6}(u, u) = 1/2$, indicating that a list element represented by u may or may not point to another list element represented by u (e.g., in Figure 5 u_2 points to u_3, but not to u_1); such an element may even point to itself.

2.2.1 Summary nodes

Nodes in a 3-valued structure that may represent more than one individual from a given 2-valued structure are called *summary nodes*. For example, in the structure shown in Figure 5, the nodes u_1, u_2, and u_3 are represented by the summary node u in Figure 6.

We use a designated unary predicate *sm* to maintain summary-node information. A summary node w has $sm^S(w) = 1/2$, indicating that it may represent more than one node from the 2-valued structure. These nodes are depicted graphically as dotted ellipses. In contrast, if $sm^S(w) = 0$, then w is known to represent a unique node. Only a node with $sm^S(w) = 1/2$ can have multiple nodes mapped to it by an embedding function.

Example 2.2 [Sorted Lists] The 3-valued structure S_o is shown in Figure 7. In contrast with structure S_6 of Figure 6, the fact that $inOrder[dle, n]^{S_o}(u_0)$ and $inOrder[dle, n]^{S_o}(u)$ are both 1 means that S_o represents all lists (with two or more elements) that are sorted in non-decreasing order according to the values of the elements' `d`-components. This illustrates how the instrumentation predicates, which have a purely local viewpoint, provide the ingredients for global properties. As will be discussed in Section 3.2, global properties can be stated via quantified formulae over the instrumentation predicates (cf. Example 3.1).

The fact that $dle^{S_o}(u_0, u)$ is 1 indicates (as would be expected) that the first list element holds the minimum of the values in the list.

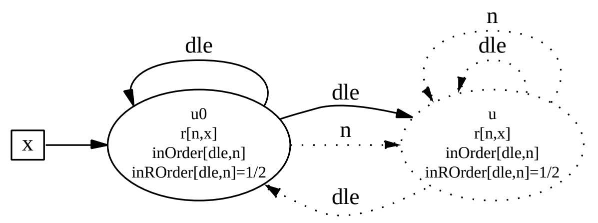
The exact choice of which nodes should be summarized is crucial for the precision of an analysis; this is discussed further in Section 3.1.

2.3 Formulae

Properties of structures can be extracted by evaluating formulae. We use first-order logic with transitive closure and equality, but without function symbols and constant symbols. For example, the formula

$$\exists v_1 : (x(v_1) \wedge n^*(v_1, v)) \tag{1}$$

extracts reachability information. Here, n^* denotes the reflexive transitive closure of the predicate n. Therefore, in every



	x	$r[n,x]$	$is[n]$	$c[n]$	$inOrder[dle,n]$	$inROrder[dle,n]$
$u0$	1	1	0	0	1	1/2
u	0	1	0	0	1	1/2

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

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

Figure 7: The 3-valued structure S_o, shown above, represents lists of length 2 or more that are pointed to by program variable `x`, and whose elements are sorted in non-decreasing order according to the values of their `d`-components.

structure S, $x(v_1)$ evaluates to 1 if v_1 is the node pointed to by `x` and $n^*(v_1,\ v)$ evaluates to 1 in S if there exists a path of zero or more `n`-edges from v_1 to v. The third column of Table 2 displays the defining formulae for all of the instrumentation predicates used in this paper.

We say that a formula φ is *potentially satisfied* on a structure S if there exists an assignment for which φ evaluates to 1 or 1/2 on S.

The Embedding Theorem: The Embedding Theorem (see [25, Theorem 3.11]) states that any formula that evaluates to a definite value in a 3-valued structure evaluates to the same value in all of the 2-valued structures embedded into that structure. The Embedding Theorem is the foundation for the use of 3-valued logic in static-analysis: It ensures that it is sensible to take formulae that—when interpreted in 2-valued logic—define the operational semantics, and reinterpret them on 3-valued structures. Formulae that define the operational semantics for programs that manipulate linked lists are given in [25, 26, 18].

For example, evaluating Formula (1) on the 3-valued structure shown in Figure 6, yields 1 for $v \mapsto u_0$, which indicates that the list element represented by u_0 is reachable from variable `x`, and 1/2 for $v \mapsto u$, which indicates that the list elements represented by u may or may not be reachable from the program variable `x`. Notice, however, that $r[n,x]^{S_6}(u) = 1$, which is more precise. This is a general principle with instrumentation predicates (referred to as the *instrumentation principle* in [25, 26]): In 3-valued structures, the stored information for an instrumentation predicate can be more precise than the result of evaluating the predicate's defining formula.

3 Proving Partial Correctness of Sorting and List-Manipulation Procedures

In this section, we describe how the 3-valued-logic analysis framework can be used to prove that an implementation of an abstract datatype (ADT) is partially correct. Here we will be concerned with an ADT of sorted linked lists—i.e., a *subset* of the full set of data structures allowed according to the C typedef shown in Figure 1, consisting of those structures that meet the "is-sorted" datatype invariant. In the case of sorted linked lists, we are interested not just in the correctness of various sorting operations, which create sorted linked lists, but also in establishing that "is-sorted" is maintained by list-manipulation operations, such as element-insertion, element-deletion, destructive list reversal, and merging of two lists.

A sorting procedure is *partially correct* if, whenever the procedure terminates, the output list it produces is sorted in non-decreasing order. Our approach to verification is capable of establishing this. For instance, the specific analysis that we discuss below establishes that at program point l_2 in Figure 3, program variable `x` always points to a list sorted in non-decreasing order (cf. Figure 9). It also establishes that at program point l_6, program variable `w`, which holds the reversal of the merge of two sorted lists, always points to a list sorted in non-increasing order (cf. Figure 12).

Verification will sometimes fail because the analysis is *conservative*, i.e., it may be that the analysis reports that, at a given program point, a variable might point to something other than a sorted list when in fact it always does point to a sorted list. Our limited experience with several small but intricate programs indicates that this does not happen. The Embedding Theorem and the usage of the abstract-interpretation methodology [7] guarantees that the converse is impossible: The analysis can never say that at a given program point l, variable `x` always points to a sorted list, and yet there is an input that leads to a store at l in which the `x` list is not sorted. Thus, if the analysis says that at the exit vertex variable `x` always points to a sorted list, then `x` will always point to a sorted list when the procedure finishes execution.

This capability should be contrasted with run-time testing, which can only show the presence of errors, not their absence [9].

As mentioned in the Introduction, an artifact of our approach is that some of the work involved in verification takes place at the level of the ADT definition, rather than at the level of an individual program or individual statements of a program. Section 3.1 discusses what is required to define a suitable analysis for observing ADT properties; Section 3.2 describes how such an analysis can then be used to check the partial correctness of ADT operations.

3.1 Specifying an Analysis for Observing ADT Properties

A set of stores that may arise before a statement st is represented by a logical structure S (i.e., an interpretation of the core and instrumentation predicates). An operational semantics of st must describe how such an interpretation is changed by the execution of st. In our work, the operational semantics is specified by a set of *predicate-update formulae* associated with each statement type (or condition)—one such formulae for each core and instrumentation predicate. For instance, suppose structure S represents a set of stores that arise before statement st. A structure S' that represents the corresponding set of stores that arise after st is obtained by evaluating the predicate-update formulae for st. (Evaluation of the formulae in 2-valued logic captures the transfer function for st of the concrete semantics; evaluation of the formulae in 3-valued logic captures the transfer function for st of the abstract semantics.) For example, suppose that q is a binary predicate, and that its predicate-update formula for statement st is $q'(v_1, v_2) = \varphi(v_1, v_2)$. The table for predicate q in S' is obtained by evaluating $\varphi(v_1, v_2)$ with each possible binding of individuals $u_1, u_2 \in U^S$ to the logical variables v_1 and v_2.

There can also be *precondition* formulae that define when an operation is allowed to be applied. The latter are used to define the effect of conditions.

Predicate-update formulae for the core predicates x (for all program variables `x`) and n, along with definitions and predicate-update formulae for the instrumentation predicates is, c, and $r[n, x]$ can be found in [25] and [18]. To define an analysis suitable for verifying procedures that operate on sorted linked lists, we have to provide predicate-update formulae for the core predicate dle and the instrumentation predicates $inOrder[dle, n]$ and $inROrder[dle, n]$ (for each of the statements that manipulate pointer variables). Updating the core-predicate dle is easy for all statement kinds except `malloc` statements; because non-`malloc` statements do not create any new individuals, the predicate-update formula is $dle'(v_1, v_2) = dle(v_1, v_2)$. For `malloc` statements, the predicate-update formula sets the dle value for newly created individuals to 1/2, i.e.,

$$dle'(v_1, v_2) = \begin{cases} 1 & new(v_1) \wedge new(v_2) \wedge v_1 = v_2 \\ dle(v_1, v_2) & v_1 = v_2 \vee \neg(new(v_1) \vee new(v_2)) \\ 1/2 & \text{otherwise} \end{cases}$$

where the predicate $new(v)$ holds for the newly allocated list element.

A trivial way to obtain safe predicate-update formulae for instrumentation predicates is to revaluate their defining formulae in the resultant structure after the core-predicates have been updated. However, this solution is almost always overly conservative since it may yield 1/2 even when the instrumentation predicate cannot be changed by the statement at all. A more precise solution is to require that the ADT designer provide a *change-formulae* $c_p^{st}(v)$ for every instrumentation predicate, identifying for which individuals u the execution of st changes $p(u)$. In the case of $inOrder[dle, n]$ and $inROrder[dle, n]$ this turns out to be quite simple because a statement st of the form `x = exp` cannot change the structure of the heap at all, and thus $c^{st}_{inOrder[dle,n]}(v) = 0$ and $c^{st}_{inROrder[dle,n]}(v) = 0$. When statement st is of the form `x->n=NULL` or `x->n=t`, only the `n` field of the node pointed to by `x` may be changed, and thus $c^{st}_{inOrder[dle,n]}(v) = x(v)$ and $c^{st}_{inROrder[dle,n]}(v) = x(v)$.

The predicate-update formulae for instrumentation predicates use the change formulae to recompute the instrumentation predicate's value only for individuals for which the predicate's value may change. Formally, we define the predicate-update formula as

$$p'(v) = c_p^{st}(v) ? \varphi_p[\{c \mapsto \varphi_c^{st} | c \in \text{CorePredicates}\}](v) : p(v)$$

where φ_p is the defining formula of instrumentation predicate p and φ_c^{st} is the predicate-update formula for core-predicate c. The formula $(\varphi_0 ? \varphi_1 : \varphi_2)$ evaluates to (i) the value of φ_1 when φ_0 evaluates to 1; (ii) the value of φ_2 when φ_0 evaluates to 0; (iii) the value of φ_1 when φ_0 evaluates to 1/2, and φ_1 and φ_2 evaluate to the same value; and (iv) 1/2 otherwise. Thus, the above formula is safe because the defining formula for $p(v)$ is recomputed for all individuals u for which the execution of st changes the value of $p(u)$.

The ADT designer is obliged to show that the change formulae are *safe*; i.e., it must not be possible for the formula to evaluate to 0 for some structure S and individual u, and yet the execution of st on S changes the value of $p(u)$. For the sorting example, this task was easy because $inOrder[dle, n]$ and $inROrder[dle, n]$ are "local" properties. However, for global properties such as reachability, the approach described above may result in an overly conservative analysis. In [25], we developed predicate-update formulae for $r[n, x]$ that define the reachability properties after a statement's execution in terms of the reachability properties that hold before the statement executes.

For conditions with no side effects, we need only write precondition formulae for the true-branches of conditions (see Table 4); the precondition formulae for the false-branches are the negations of these formulae.

Condition	**Precondition formula for true-branch**
`x == y`	$\exists v : x(v) \wedge y(v)$
`x != y`	$\neg\exists v : x(v) \wedge y(v)$
`x == NULL`	$\neg\exists v : x(v)$
`x != NULL`	$\exists v : x(v)$
`x->d <= y->d`	$\exists v_1, v_2 : x(v_1) \wedge y(v_2) \wedge dle(v_1, v_2)$
`x->d == y->d`	$\exists v_1, v_2 : x(v_1) \wedge y(v_2) \wedge dle(v_1, v_2) \wedge dle(v_2, v_1)$
`x->d < y->d`	$\exists v_1, v_2 : x(v_1) \wedge y(v_2) \wedge \neg dle(v_2, v_1)$
uninterpreted	1/2

Table 4: Precondition formulae for the true-branch part of atomic program conditions that manipulate linked lists.

The transformers so defined are used in an iterative algorithm to compute, for each program point (control-flow graph node), a finite set of 3-valued structures that conservatively represent the set of stores that can possibly occur at that point.

The abstraction function of the static analysis is defined by a subset of the unary predicates, which we call the *abstraction properties*. The principle behind abstraction is that all list elements that have the same values for the abstraction predicates are mapped to the same abstract element. Thus, if we view the set of abstraction predicates as our means of observing the contents of the heap, the heap cells summarized by one summary node are those that have no observable differences.

Note that for a fixed set of abstraction properties, there can only be a constant number of nodes with observable differences. Thus, to guarantee that the analysis terminates for procedures with loops, the iterative fixed-point finding procedure keeps collapsing the structures that arise, by merging all nodes in a given structure that have no observable differences. By this means, the number of nodes in any 3-valued structure is *bounded*, and the analysis must eventually terminate.

(The above summary glosses over several important details needed for boosting the precision of the technique. Details can be found in [25].)

3.2 Specifying and Checking Partial Correctness of ADT Operations

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

- The procedure's control-flow graph.
- A set of 3-valued structures that characterize the acceptable inputs to the procedure.
- Formulae that characterize the acceptable outputs of a correctly working procedure.

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 formulae that characterize the acceptable outputs are evaluated on the structures that are generated by the analysis at the procedure's exit point.

Example 3.1 Consider the problem of establishing that the version of `insert_sort` shown in Figure 2 is partially correct. Figure 8 shows the three structures that characterize the set of stores in which program variable `x` points to an acyclic, unshared linked list.[3] After running the analysis of `insert_sort`, 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:

$$\forall v : r[n,x](v) \Rightarrow \mathit{inOrder}[\mathit{dle},n](v). \tag{2}$$

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

However, at this point, the reader may smell a rat: A "sorting" procedure that always returns `NULL` will satisfy Formula (2) at the exit point! Thus, Formula (2) is only part of the specification of the post-condition of a correct sorting procedure. A second property required of a correct sorting procedure (as well as of many other procedures that manipulate sorted linked lists) is that the output list must be a permutation of the input list.

We can establish that the permutation property holds for the output of `insert_sort` by extending the program-analysis specification with another predicate, $\mathit{orig}[n,x](v)$, whose value is set at the entry point to record the elements that are reachable from `x` there. In each statement, the predicate-update formula used for $\mathit{orig}[n,x]$ is $\mathit{orig}'[n,x](v) = \mathit{orig}[n,x](v)$. In other words, $\mathit{orig}[n,x]$ serves as an indelible mark on the elements initially reachable from `x`. At the end of the procedure, we then need to check that the following formula evaluates to 1:

$$\forall v : \mathit{orig}[n,x](v) \Leftrightarrow r[n,x](v). \tag{3}$$

If the formula does evaluate 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.

In this case, predicate $\mathit{orig}[n,x](v)$ has been introduced to make it possible to observe whether the output list is a permutation of the input list. In general, a predicate of this kind is similar to an auxiliary variable of the kind often used in conventional program verification to denote the initial value of a program variable [13, Section 6.2].

[3] These are exactly the 3-valued structures that the analysis discovers as the possible outputs of `create`.

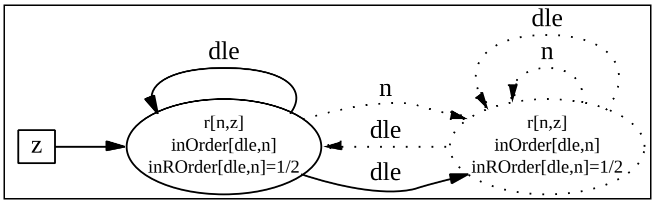


Figure 11: A structure that arises at l_5.

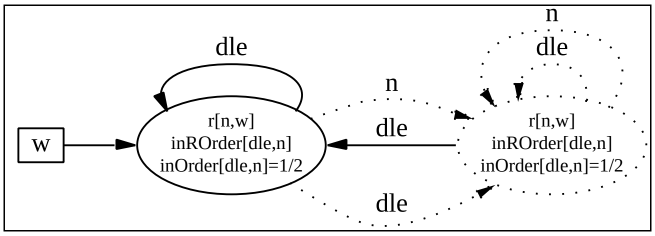


Figure 12: A structure that arises at l_6.

Example 3.2 Figure 9 shows the three 3-valued structures that arise at the end of `insert_sort`, given the structures shown in Figure 8 as the input structures. The structures in Figure 9 describe all possible stores in which variable `x` points to an acyclic, unshared, *sorted* linked list. In all three structures, Formulas (2) and (3) both evaluate to 1.[4] Consequently, `insert_sort` is guaranteed to work correctly on all acceptable inputs.

Example 3.3 Figure 10 shows one of the structures that can arise at program point l_4 of `main` (see Figure 3). In Figure 10, the substructure consisting of the `x`-box together with the upper two nodes represents one sorted list of length 2 or more; the `y`-box and the lower two nodes represents a second sorted list of length 2 or more.

Figure 11 shows what is produced by the analysis when the structure shown in Figure 10 is supplied as the input structure in the analysis of `merge`: The output structure represents the acyclic, unshared, *sorted* linked lists of length 2 or more. In other words, `merge` preserves sortedness.

Figure 12 shows what is produced when the structure shown in Figure 11 is supplied as the input structure in the analysis of `reverse`: The output structure represents the acyclic, unshared, linked lists of length 2 or more, *sorted in reverse order*.

Overall, the method described above is able to establish that at program point l_6 of `main` (Figure 3), program variable `w`—which is computed by reversing a list created by merging two sorted lists—always points to a list sorted in non-increasing order.

4 Compile-Time Debugging of Programs

As observed earlier, our technique will never report that the lists produced by a sorting program are sorted when, in fact, there is some input that leads to unsorted output. In this section, we demonstrate how the output of the algorithm when applied to incorrect programs provides information that is useful for catching bugs at compile time.

[4] Assuming, in the case of Formula (3), that instrumentation predicate $\mathit{orig}[n,x]$ was added to the analysis.

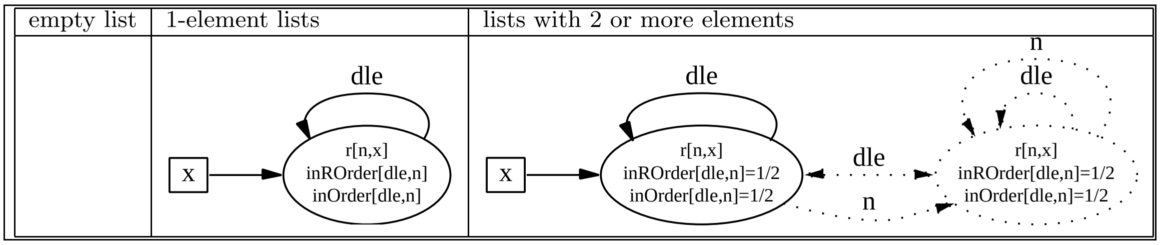


Figure 8: The structures that arise at l_1.

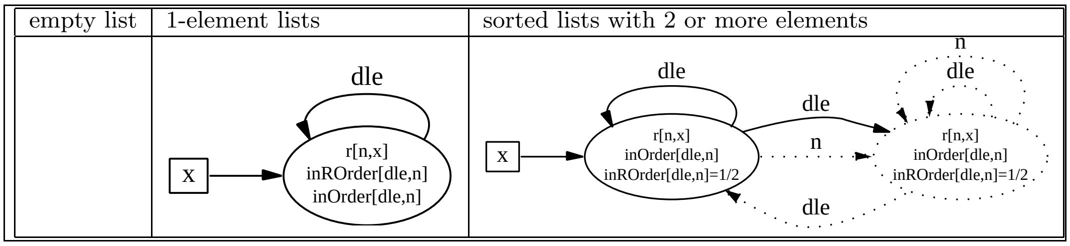


Figure 9: The structures that arise at l_2.

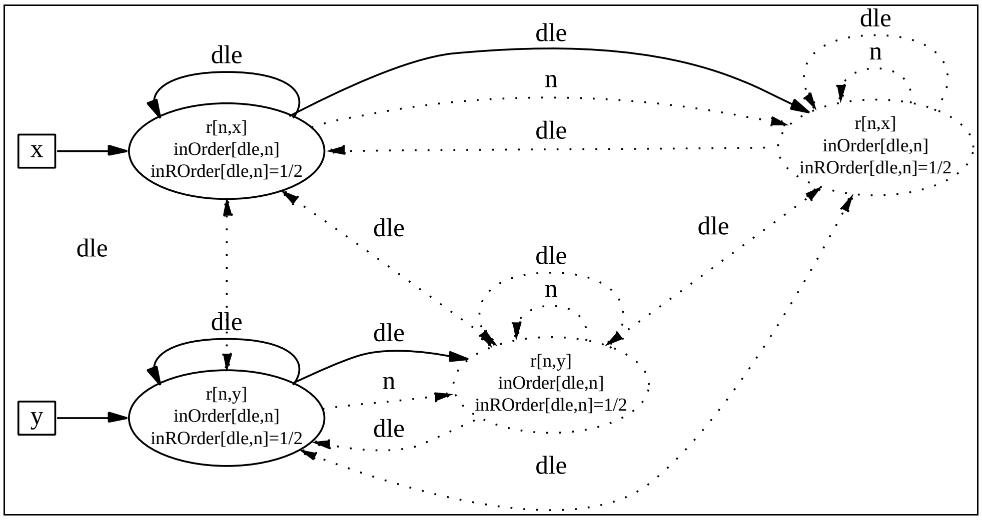


Figure 10: A structure that arises at l_4.

A common error in sorting programs is to forget to make comparisons on boundary elements of the input data [22]. Another error reported in [22] is that the output list is not sorted in the specified order. When our verification method fails to confirm that a program is a correct sorting procedure, such bugs can sometimes be diagnosed by inspecting the 3-valued structures that have been produced by the analysis.

Example 4.1 Figure 13 shows an incorrect version of insertion sort that ignores the first element. A structure that arises after the program's exit point is shown in Figure 14. This structure provides the following clues about the nature of the bug:

- Because $inOrder[dle, n]$ is 1 for the summary node, most of the elements of the list must be arranged in non-decreasing order.
- However, $inOrder[dle, n]$ is 1/2 for the first node, which indicates that it may or may not be in the proper place with respect to the rest of the list elements.

Example 4.2 Figure 15 shows a bubble-sort procedure. For the analysis of this procedure, we added an instrumentation predicate, $DataIsNEqual[dle, n]$, defined by the formula:

$$\forall v_1 : n(v, v_1) \Rightarrow \neg(dle(v, v_1) \wedge dle(v_1, v)).$$

A subtle bug in bubble-sort arises if we change the condition for swapping elements from "`y->data > yn->data`" to "`y->data >= yn->data`". With this change, the procedure does not terminate if the input list contains two elements whose `d`-fields have identical values. The reason is that in every pass over the list, these two elements would be swapped, and thus the sort would never terminate. Because such a bug only manifests itself for certain inputs, this is something that testing could overlook.

When our verification method is applied to the erroneous bubble-sort program, and supplied with the input structures from Figure 8 (i.e., arbitrary acyclic, unshared linked lists),

```
#include "list.h"
L insert_sort_b2(L x) {
  L r, pr, rn, l, pl;
  if (x == NULL)
    return NULL;
  pr = x;
  r = x->n;
  while (r != NULL) {
     pl = x;
     rn = r->n;
     l = x->n;
     while(l != r) {
        if(l->d > r->d) {
          pr->n = rn;
          r->n = l ;
          pl->n = r;
          r = pr;
          break;
        }
        pl = l;
        l = l->n;
     }
     pr = r;
     r = rn;
  }
  return x;
}
```

Figure 13: An incorrect version of insertion sort that ignores the first element.

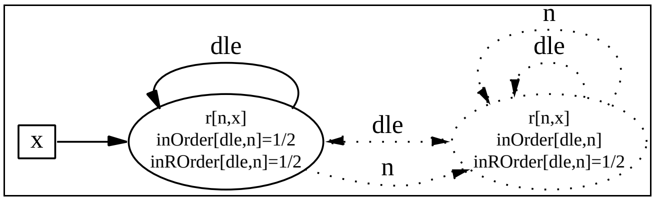


Figure 14: A structure that arises at the exit point of `insert_sort_b2`.

the analysis discovers that the only lists that can arise at the end of the procedure are ones in which all of the data fields have different values. This shows up in the final 3-valued structure as DataIsNEqual[dle, n] having the value 1 for all list elements.

Example 4.3 Figure 16 shows an incorrect version of insertion sort. Our method would indicate that Formula (3) fails to evaluate to 1 in the 3-valued structures that arise at the exit point, and thus fails to confirm that the program performs a permutation of the list (because some elements may be lost).

Additional information about the program can be obtained by checking for memory-cleanness violations, along the lines of [10].[5] By applying appropriate cleanness-checking formulae to the 3-valued structures that arise in the analysis of this program, it is possible to detect that a memory leak occurs at program point l.

[5] The method presented in this paper can avoid some of the false alarms that would be reported by the method described in [10].

```
/* bubble_sort.c */
#include "Bool.h"
#include "list.h"
void bubble_sort(L x) {
   change = TRUE;
   while (change) {
      p = NULL;
      change = FALSE;
      y = x;
      yn = y->n;
      while (yn != NULL) {
         if (y->data > yn->data) {
            t = yn->n;
            change = TRUE;
            y->n = t;
            yn->n = y;
            if (p == NULL)
               x = yn;
            else
               p->n = yn;
            p = yn;
            yn = t;
         } else {
            p = y;
            y = yn;
            yn = y->n;
         }
      }
   }
}
```

Figure 15: A correct version of bubble sort.

5 A Prototype Implementation

Section 2 reviewed the method for creating program-analysis algorithms described in [25, 26]. This approach provides a *parametric* framework for program analysis. That is, it provides a method for generating different program-analysis tools from high-level, user-supplied descriptions of what is desired. The ideal is to have a fully automatic method—a *yacc* for program analysis, so to speak. A prototype version of such a system, called TVLA, has recently been implemented in Java [18, 19]. TVLA mostly spares the user from having to possess a deep knowledge of program analysis (just as *yacc*, by generating a parser from a context-free grammar, spares a *yacc* user from having to understand the details of LALR(1) parsing theory).

The current implementation of TVLA is capable of handling programs that use pointers, including allocation statements, deallocation statements, and destructive updates through pointers. Casting and pointer arithmetic is not supported. Currently TVLA supports only intraprocedural analysis.

The analysis described in this paper has been implemented in TVLA. Because TVLA does not currently support interprocedural analysis, we have conducted "unit tests" by analyzing one procedure at a time on the relevant input structures. Performance information for the analysis algorithm running on a Pentium II 400 MHz under Linux with JDK 1.2 is presented in Table 5.[6] (Note that the figure reported for "Number of Structures" is the total number of structures that were cre-

[6] Our experience has been that the system runs about 20% faster using JVM under Windows.

```
#include "list.h"
L insert_sort_b1(L x) {
  L r, pr, rn, l, pl;
  r = x;
  pr = NULL;
  while (r != NULL) {
     l = x;
     pl = NULL;
     while (l != r) {
        if (l->data > r->data) {
          pr->n = r->n;
          r->n = l;
          if (pl == NULL)
            x = r;
          else
            pl->n = r;
          r = pr;
          break;
        }
        pl = l;
        l = l->n;
     }
     pr = r;
     l: r = r->n; /* creates a leak */
  }
  return x;
}
```

Figure 16: An incorrect version of insertion sort, in which a memory leak occurs at program point l.

Procedure	**Number of Structures**	**Time (seconds)**
create	9	0.45
insert_sort	2963	158.695
merge	1238	74.092
reverse	87	2.266
insert_b1	8198	627.309
insert_b2	1823	114.474
bubble_sort	4350	245.74
bubble_sort_b	4794	295.338

Table 5: Running time and total number of structures that arise during analysis for the procedures analyzed.

ated at all program points, not the number of structures kept at an individual control-flow-graph node.)

We have also used TVLA to implement a version of the analysis that works on dynamic arrays. This is actually somewhat simpler than what has been described above because the cyclicity and sharing instrumentation predicates are not needed.

6 Limitations, Related Work, and Future Directions

So far, the implementation is only able to analyze small programs in a "friendly" subset of C. We will try to extend it to a larger subset of C, and to scale it up to programs of realistic size. One possible way involves first running a cheap and imprecise pointer-analysis algorithm, such as the flow-insensitive points-to analysis described in [27], before proceeding to our quite precise but expensive analysis. We will also look into reducing the storage requirements of our method by representing the predicate tables with BDDs [2].

Verification systems such as [16] have an advantage over our method because they compute weakest preconditions. Information is propagated backwards, which may lead to a significant storage savings; however, as noted in the Introduction, verification systems require that loop invariants be supplied by the user. For erroneous programs, verification systems that use weakest preconditions can also produce counterexamples, which provides useful feedback to the programmer. However, the system in [16] cannot be used to establish the correctness of sorting algorithms, and is significantly slower than TVLA, even for proving memory-cleanness properties. One possible explanation is that propagating formulae is more expensive than propagating structures. It is conceivable that our method could be combined with demand dataflow analysis to propagate information backwards; however, the presence of destructive updates greatly complicates the problem.

Only intraprocedural analysis is currently supported in TVLA. Therefore, the analysis of a recursive version of quicksort is not yet possible.[7] An extension to perform interprocedural analysis is under development.

In general, our analysis cannot establish liveness properties, e.g., that the invocation of a function always terminates. However, in many cases, it is possible to establish the negation of a liveness property by checking a safety property. For example, we used our analysis to show that when the `change` flag is not set in bubble sort, the program never terminates. In Section 4, we discussed a more interesting example in which our analysis detected that a buggy version of bubble sort does not terminate on a list that contains some elements with equal values.

Even for safety properties, however, our technique can sometimes be overly imprecise. For example, the set of instrumentation predicates discussed in Section 3 does not allow us to conclude that a sorting program is *stable*, i.e., never reorders two elements with the same `d`-field.

Other abstract-interpretation algorithms can also be used for verification. For instance, in [1], it is shown that intervals can be used to precisely analyze McCarthy's "91-function". There it is also shown how to propagate information in the backwards direction. However, intervals do not provide sufficient information to handle dynamically allocated data structures and destructive updates or even the sortedness properties of arrays.

Abstract interpretation has also been used for verification in Cecil/Cesar [23, 24] and FLAVERS [11]. However, the kind of verification performed in that work is of a different character from what we have done in the present paper. Both Cecil/Cesar and FLAVERS address the problem of checking whether the events in the execution of a program always meet a specified sequencing constraint. In contrast, the goal of our work is to verify partial correctness in the sense of Hoare [14]. Because our work addresses the use of *dynamically allocated* heap-allocated storage—and hence must characterize an unbounded number of storage elements—it concerns a situation

[7] We have tried running the algorithm on a version of quicksort that uses an explicit stack, but the algorithm ran out of space. It is possible that this may be overcome by using extra instrumentation predicates to track relationships between stack and list elements. Another plausible approach would be to use an inductive strategy—using an assertion that a recursive call returns a sorted list. Our emphasis to date has been on purely automatic analyses, where the only user-supplied assertion required is one that characterizes the input to the procedure.

that has always been considered quite difficult in the verification community [21]. A surprising fact about our work is that it shows that, in some cases, the problem can be addressed by means of static program analysis. Our work also addresses pointer indirections and aliases, which are especially difficult to handle when heap-allocated elements are considered.

Model checking [5] and program analysis are two different techniques that can be used to establish safety properties of programs. In some ways, however, the TVLA system "feels" very similar to SMV [20]; in both systems, the operational semantics is specified as a transition relation. On the other hand, the use of first-order logic in TVLA enables it to handle dynamically allocated objects, which are outside the scope of current model-checking techniques. Furthermore, the use of 3-valued logic (and abstraction with respect to a subset of the unary predicates) allows TVLA to automatically obtain a transition relation that operates on bounded-size representations, which ensures that the analysis terminates. However, because most model-checking techniques are based on propositional logics and finite automata, many optimization techniques are possible, such as the use of BDDs to represent sets of propositions.

The original problem that motivated the use of 3-valued logic for program analysis was *shape analysis*, where the goal is to determine, for each program point, a set of "shape descriptors" that characterize the possible shapes of the structures that could have been constructed using dynamically allocated storage [25, 26]. In this paper, the machinery for performing static program analysis via 3-valued logic was applied to a quite different problem, namely that of characterizing the intermediate and final states of sorting algorithms.

Our use of shape analysis for verification is different from the use of shape analysis by Corbett [6].[8] Corbett uses the results of shape analysis to improve the performance of a model checker for concurrent Java programs. Shape analysis is used to identify which heap-allocated variables are purely local to one thread and which shared variables are protected by locks. This information is used, in turn, to avoid having to explore all possible interleavings of threads.

The fact that the verification method proposed in this paper is based on the method from [25, 26] for creating program-analysis algorithms has advantages and disadvantages: Although the developer of a program-analysis algorithm is freed from the proof obligations normally associated with abstract interpretation, he is left with the responsibility of showing that the predicate-update formulae correctly reflect how predicates are affected by the execution of the various kinds of statements in the programming language. There are several reasons for believing that this is much less of a burden than that normally imposed with standard approaches to abstract interpretation:

- As pointed out in the Introduction, the formulae (and associated proofs) are actually not for a specific *program*, but for a particular *datatype*—once designed, they can be used in proving the correctness of multiple programs that manipulate data of that type. For instance, in this paper, we were able to adopt unchanged all of the predicates used for shape analysis of linked-list programs [25, 26]. As more experience is gained with this approach, libraries of predicate-update formulae (with standard proofs of correctness) will be developed.
- In Section 3.1, we described an approach to creating families of predicate-update formulae in terms of *change-formulae*. Factoring the predicate-update formulae in this way makes it much easier to provide the necessary correctness proofs.
- We have some ideas about how to automatically generate correct predicate-update formulae for the instrumentation predicates.

[8]Corbett uses a version of the shape-analysis algorithm described in [3].

References

[1] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, New York, NY, 1993. ACM Press.

[2] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, C-35(6):677–691, August 1986.

[3] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.

[4] T.E. Cheatham, Jr., G.H. Holloway, and J.A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Trans. on Softw. Eng.*, 5(4):402–417, 1979.

[5] E.M. Clarke, Jr., O. Grumberg, and D.A. Peled. *Model Checking.* The M.I.T. Press, 1999.

[6] J.C. Corbett. Using shape analysis to reduce finite-state models of concurrent Java programs. *Trans. on Softw. Eng. and Method.*, 9(1):51–93, 2000.

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[8] L.P. Deutsch. *An Interactive Program Verifier.* PhD thesis, Univ. of California, Berkeley, CA, 1973.

[9] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[10] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Static Analysis Symp.*, 2000.

[11] M.B. Dwyer and L.A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proc. of the 2nd ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, pages 62–75, New York, NY, 1994. ACM Press.

[12] W. Gillett. *Iterative Global Flow Techniques for Detecting Program Anomalies.* PhD thesis, Univ. of Illinois, 1977.

[13] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[14] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 583, October 1969.

[15] C.A.R. Hoare. Recursive data structures. *Int. J. of Comp. and Inf. Sci.*, 4(2):105–132, 1975.

[16] J.L. Jensen, M.E. Joergensen, N.Klarlund, and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 1997.

[17] J.C. King. *A Program Verifier.* PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA, 1969.

[18] T. Lev-Ami. TVLA: A framework for Kleene based static analysis. Master's thesis, Tel-Aviv University, 2000. Available at http://www.math.tau.ac.il/∼tla.

[19] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, 2000.

[20] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[21] J.M. Morris. Assignment and linked data structures. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 35–41. D. Reidel Publishing Co., Boston, MA, 1982.

[22] G.J. Myers. *The Art of Software Testing.* John Wiley and Sons, 1978.

[23] K.M. Olender and L.J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Trans. on Softw. Eng.*, 16(3):268–280, 1990.

[24] K.M. Olender and L.J. Osterweil. Interprocedural static analysis of sequencing constraints. *Trans. on Softw. Eng. and Method.*, 1(1):21–52, January 1992.

[25] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. Tech. Rep. TR-1383, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, August 1998. (Revised March 2000.) Available at "http://www.cs.wisc.edu/wpis/papers/parametric.ps".

[26] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999.

[27] B. Steensgaard. Points-to analysis in almost-linear time. In *Symp. on Princ. of Prog. Lang.*, pages 32–41, 1996.

[28] N. Suzuki and K. Ishihata. Implementation of array bound checker. In *Symp. on Princ. of Prog. Lang.*, pages 132–143, New York, NY, 1977. ACM Press.

[29] M. Tamir. ADI: Automatic derivation of invariants. *IEEE Trans. on Softw. Eng.*, 6(1):40–48, 1990.

[30] B. Wegbreit. The synthesis of loop predicates. *Commun. ACM*, 17(2):102–112, 1974.

A Auxiliary Procedures Analyzed

Procedures `reverse`, `create`, and `merge` are given in Figures 17, 18, and 19, respectively.

```
#include "list.h"
List reverse(List x) {
  List y, t;
  y = NULL;
  while (x != NULL) {
    t = y;
    y = x;
    x = x->n;
    y->n = t;
  }
  t = NULL;
  return y;
}
```

Figure 17: A C procedure that reverses a list (using destructive updating).

```
#include "list.h"
L create() {
  int i, rand(void);
  L temp, r;
  char *malloc(int);
  i = rand();
  r = NULL;
  while (i>0) {
     temp = (L)malloc(sizeof(struct node));
     temp->d = rand();
     temp->n = r;
     r = temp;
     i--;
  }
  return r;
}
```

Figure 18: A C procedure that creates a random list of positive integers.

```
#include "list.h"
L merge(L p, L q) {
  L head_list, tail_list;
  if (p==NULL) return q;
  if (q==NULL) return p;
  if (p->d < q->d) {
   head_list = p;
   p = p->n;
  } else {
   head_list = q;
   q = q->n;
  }
  tail_list = head_list;
  while ((p!=NULL) && (q!=NULL)) {
     if (p->d < q->d) {
       tail_list->n = p;
       p = p->n;
     } else {
       tail_list->n = q;
       q = q->n;
     }
     tail_list = tail_list->n;
  }
  if (p!=NULL)
   tail_list->n = p;
  else if (q!=NULL)
   tail_list->n = q;
  return head_list;
}
```

Figure 19: A C procedure that merges two sorted lists.