TVLA: A System for Implementing Static Analyses*

Tal Lev-Ami and Mooly Sagiv

Department of Computer Science, Tel-Aviv University, Israel
{tla,sagiv}@math.tau.ac.il

Abstract. We present TVLA (Three-Valued-Logic Analyzer). TVLA is a "YACC"-like framework for automatically constructing static-analysis algorithms from an operational semantics, where the operational semantics is specified using logical formulae. TVLA has been implemented in Java and was successfully used to perform shape analysis on programs manipulating linked data structures (singly and doubly linked lists), to prove safety properties of Mobile Ambients, and to verify the partial correctness of several sorting programs.

1 Introduction

The abstract-interpretation technique [5] for static analysis allows one to summarize the behavior of a statement on an infinite set of possible memory states. This is sometimes called an *abstract semantics* for the statement. With this methodology it is necessary to show that the abstract semantics is *conservative*, i.e., it summarizes the (*concrete*) *operational semantics* of the statement for every possible memory state. Intuitively speaking, the operational semantics of a statement is a formal definition of an interpreter for this statement. This operational semantics is usually quite natural. However, designing and implementing sound and reasonably precise abstract semantics is quite cumbersome (the best induced abstract semantics defined in [5] is usually not computable). This is particularly true in problems like shape analysis and pointer analysis (e.g., see [6, 17, 15]), where the operational semantics involves destructive memory updates.

In this paper, we present TVLA (**T**hree-**V**alued-**L**ogic **A**nalyzer), a system for automatically generating a static-analysis algorithm from the operational semantics of a given program. The operational semantics is written in a special form, based on first-order predicate logic with transitive closure. An additional input to TVLA is an abstract representation of all the possible memory states at the beginning of the analyzed program. TVLA automatically generates the abstract semantics, and, for each program point, produces a conservative abstract representation of the memory states at that point.

* Supported, in part, by a grant from the Academy of Science, Israel.

1.1 Main Results

TVLA is intended as a proof of concept for intra-procedural shape analysis, and other static-analysis algorithms. It is a test-bed in which it is quite easy to try out new ideas. The theory behind TVLA is based on [16, 17] (see Sect. 5.2). The system is publicly available from `http://www.math.tau.ac.il/~tla`.

TVLA was implemented in Java and has been successfully used to perform shape analysis on programs manipulating linked data structures (singly and doubly linked lists), to prove safety properties of Mobile Ambients, and to verify partial correctness of several programs. We also report on some programs that are too complex for the current system. The system was tested on a Pentium II 400 MHz running Linux with JDK 1.2. All the timing information about the system refers to this computer[1].

Applications TVLA has been utilized to analyze a variety of small but intricate programs from the groups described below.

Singly Linked Lists. We performed shape analysis on the set of programs manipulating singly linked lists used in [7], including ones for searching, element insertion, and element deletion. These programs perform destructive updating. Some of these programs are (deliberately) semantically incorrect, and we are able to locate the bugs in them. The analysis times are reported in AppendixA.

Doubly Linked Lists. Doubly linked lists are more challenging than singly linked lists because they create shared memory cells and cycles. We have analyzed a program that inserts a new element into an arbitrary place in a doubly linked list, and the analysis was able to conclude that the insertion results in a doubly linked list.

Sorting Programs. A different kind of application of TVLA is for program verification (see [11]). We applied TVLA to several implementations of sorting algorithms, and proved that, given a possibly unsorted linked list as input, we always end up with a sorted list. This is proven without the need for programmer-specified loop invariants. Instead, the operational semantics also keeps track of inequalities between the list elements. We are encouraged by the fact that we have successfully verified both insert sort and bubble sort on singly linked lists.

Mobile Ambients. We implemented the analysis of [13] and found out that it is imprecise and quite slow. This motivated us to generalize the techniques presented in [16, 17] in order to guarantee that only a constant number of structures arise at each program point (see Sect. 3.4). With this extension, TVLA was able to successfully analyze a slight variant of the original specification used in [13]. This took 336 CPU seconds and the analysis proved the necessary properties (uniqueness of ambient instance and mutual exclusion) precisely.

[1] Our experience indicates that using JVM on Windows, the system runs about 20% faster.

1.2 Outline of the Paper

The rest of the paper is organized as follows. In Sect. 2, we give a primer on the use of 3-valued logic in static analysis. Sect. 3 contains an overview of the TVLA system and its capabilities. Sect. 4 gives a description of the analyses done with the system. We conclude by summarizing related work and further research directions (Sect. 5). Appendix A presents the empirical results for test runs of the system. Appendix B presents an operational semantics for statements manipulating nodes of singly linked lists. For other aspects of TVLA, including algorithms, proofs, description of other features, additional examples, and a user's manual, we refer the reader to [10].

A program that destructively reverses a singly linked list is shown in Fig. 1. The shape analysis of this program serves as a running example in this paper.

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

(a)

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

(b)

Fig. 1. (a) Declaration of a linked-list data type in C. (b) A C function that uses destructive updates to reverse the list pointed to by parameter x.

2 A Primer on 3-Valued-Logic-Based Analysis

Kleene's 3-valued logic is an extension of ordinary 2-valued logic with the special value of 1/2 (unknown) for cases that can be either 1 or 0. Kleene's interpretation of the propositional operators is given in Table 1. We say that the values 0 and 1 are *definite values* and that 1/2 is an *indefinite value.*

Table 1. Kleene's 3-valued interpretation of the propositional operators.

$\wedge$	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

2.1 Representing Memory States via Logical Structures

Our vocabulary includes a set of predicate symbols partitioned into two disjoint sets: *core* and *instrumentation* predicates. Instrumentation predicates are used to observe derived properties based on core predicates.

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 (core and instrumentation) predicate p of arity k, p^S is a function $p^S : (U^S)^k \to \{0, 1\}$. 2-valued structures are used to represent memory states used in the operational semantics of the program.

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.)

TVLA only supports predicates of arity ≤ 2; such logical structures can be thought of as directed graphs. A directed edge labeled by p from u_1 to u_2 denotes that $p^S(u_1, u_2) = 1$. Also, we draw p inside a node u when $p^S(u) = 1$.

Table 2. The core predicates used in the analysis of the running example.

Predicate	**Intended Meaning**
$x(v)$	Is v pointed to by variable `x`?
$y(v)$	Is v pointed to by variable `y`?
$t(v)$	Is v pointed to by variable `t`?
$n(v_1, v_2)$	Does the n-field of v_1 point to v_2?

Example 21 In the running example, 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 2, and the intended meaning of the instrumentation predicates is given in Table 3 (for the moment ignore the third column). The store in Fig. 2 is represented by the 2-valued structure S_3 shown in Fig. 3. The structure S_3 has four nodes, u_0, u_1, u_2, and u_3 representing the four list elements. This representation intentionally ignores the values of the data field, which are usually immaterial for the analysis.

Table 3. The instrumentation predicates used in the analysis of the running example and their meaning. Similar instrumentation predicates are used in all of our shape analyses for singly linked lists. The defining formulae are explained in Sect. 2.3.

Predicate	Intended Meaning	Defining Formula
$r[n,x](v)$	Is v reachable from program variable x using field n?	$\exists v_1 : (x(v_1) \land n^*(v_1, v))$
$r[n,y](v)$	Is v reachable from program variable y using field n?	$\exists v_1 : (y(v_1) \land n^*(v_1, v))$
$r[n,t](v)$	Is v reachable from program variable t using field n?	$\exists v_1 : (t(v_1) \land n^*(v_1, v))$
$c[n](v)$	Does v reside on a directed cycle via dereferences along n-fields?	$n^+(v, v)$
$is[n](v)$	Is v pointed to by more than one n-field	$\exists v_1, v_2 : n(v_1, v) \land n(v_2, v) \land v_1 \neq v_2$

Pointer variables are represented by unary predicates (i.e., $x^S(u) = 1$ if the variable x points to the list element represented by u). In Fig. 3, the variable x is represented by the unary predicate x, which is 1 only for u_0. Notice that TVLA allows the user to specify that a unary predicate is drawn as a box with an arrow into each node for which it holds. In Fig. 3, x is drawn as a box and has an arrow to u_0. Pointer fields within the list elements are represented as binary predicates (i.e., $n^S(u_1,\ u_2) = 1$ if the n-field of u_1 points to u_2).

The instrumentation predicate $r[n,x]$ holds for list elements that are reachable from program variable x, possibly using a sequence of accesses through the n-field. The structure S_3 in Fig. 3 has $r[n,x]^{S_3}$ set to 1 for all the nodes because they are all reachable from x. An important aspect of explicitly storing $r[n,x]$ is that we can incrementally compute the appropriate values for the predicates after execution of the program statement (see [17, Sect. 6.1]). For example, for the statement **y = x**, the nodes reachable from y after the statement executes are the same as the nodes reachable from x.

The instrumentation predicate $is[n]$ holds for nodes shared by n-fields (a node is *shared* by n-fields, if it is pointed to by more than one list element using the field n). In Fig. 3, all the elements of the list are unshared, and thus $is[n]^{S_3}$ is 0 for all of them.

The instrumentation predicate $c[n]$ holds for nodes on a cycle of accesses along n-fields. We use the cyclicity instrumentation to avoid performing a transitive-closure operation when updating the reachability information. In Fig. 3, the list is acyclic, and thus $c[n]^{S_3}$ is 0 for all of the nodes.

In fact, throughout the analysis of the running example, $is[n]^S$ and $c[n]^S$ are 0 for all of the nodes.

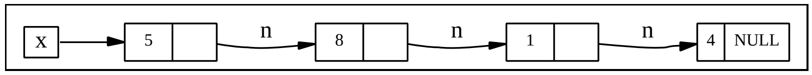

Fig. 2. A possible store for the running example.

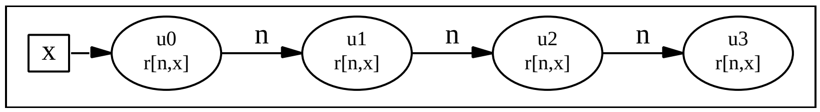

Fig. 3. A logical structure S_3 representing the store shown in Fig. 2 in a graphical representation.

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 p^S for every predicate symbol p. But, for every predicate p of arity k, p^S is a function $p^S : (U^S)^k \to \{0, 1, 1/2\}$, 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. Unary indefinite predicate values are drawn inside the nodes and marked as indefinite (this does not occur in the running example).

Let $S^\natural$ be a 2-valued structure, S be a 3-valued structure, and $f : U^{S^\natural} \to U^S$ such that f is surjective. 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 with some function f.* Thus, S can compactly represent many structures.

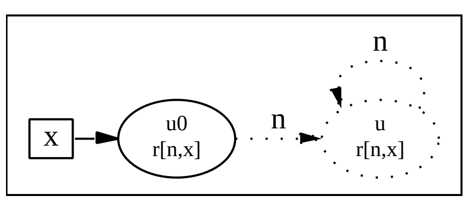

Fig. 4. A 3-valued structure S_4 representing lists of length 2 or more that are pointed to by program variable x (e.g., S_3).

Example 22 In the running example, the 3-valued structure S_4 shown in Fig. 4 represents the 2-valued structure S_3 for $f(u_0) = u_0$ and $f(u_1) = f(u_2) = f(u_3) = u$. In fact, the structure shown in Fig. 4 represents all the lists with two or more elements.

The unary predicate symbol x has $x^{S_4}(u_0) = 1$, indicating that the program variable x is known to point to the list element represented by u_0, and $x^{S_4}(u) = 0$, indicating that x is known not to point to any of the list elements represented by u.

The binary predicate symbol n has $n^{S_4}(u_0, u) = 1/2$, indicating that the n-field of the list element represented by u_0 may point to a list element represented by u — namely the second list element (u_1 in Fig. 3) — but does not point to all the list elements represented by u (e.g. u_2 in Fig. 3). Also, $n^{S_4}(u, u) = 1/2$, indicating that the n-field of a list element represented by u may point to another list element represented by u or even to itself but does not point to all the list elements represented by u (e.g., in Fig. 3 the n-field of u_2 points to u_3, but not to u_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 Fig. 3, the nodes u_1, u_2, and u_3 are represented by the single node u in Fig. 4.

TVLA uses a special designated unary predicate sm to maintain summary-node information. Such a summary node w has $sm^S(w) = 1/2$, indicating that it may represent more than one node in the embedded 2-valued structures. These nodes are graphically drawn as dotted ellipsis. In contrast, if $sm^S(w) = 0$ then w is known to represent a unique node. Only nodes with $sm^S(w) = 1/2$ can have more than one node mapped to them by the embedding function.

The exact choice of which nodes should be summarized is crucial for the precision of the analysis and is discussed in Sect. 3.2.

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 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 3 displays the defining formula of all the instrumentation predicates used in the running example.

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

The Embedding Theorem. The Embedding Theorem (see [16, Theorem 3.7]) states that any formula that evaluates to a definite value in a 3-valued structure evaluates to the same value in all 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 reinterpret on the 3-valued structures the formulae, that when interpreted in 2-valued logic, define the operational semantics.

TVLA requires each instrumentation predicate to be associated with a formula over the core predicates defining its meaning. For example, evaluating formula (1) on the 3-valued structure shown in Fig. 4, 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 program variable `x`. Notice that $r[n,x]^{S_4}(u) = 1$, which is more precise. This is a general principle with instrumentation predicates (referred to as the *instrumentation principle* in [16]). The stored information can be more precise than the result of evaluating the corresponding formula.

3 System Description

The input to TVLA consists of two files: (i) a TVS (Three Valued logical Structure) file containing a textual representation of the input structures (see Fig. 5), and (ii) a TVP (Three Valued Program) file, which includes the operational semantics and the association of the operational semantics with the edges of the control flow graph (CFG) of the analyzed program (see Figs. 6 and 7). To simplify the specification, we allow the operational semantics to be specific to the analyzed data type (e.g., singly linked lists in the running example). In the conversion of a C program into a TVP file, some normalizing transformations are applied (see [4, 15]). For example, the assignment `y->n=t` is broken into two statements: (i) `y->n=NULL`, followed by (ii) `y->n=t` assuming that `y->n==NULL`. The full operational semantics for programs manipulating singly-linked-lists of type `L` is given in Appendix B.

$$
\begin{array}{l}
\%n = \{u, u0\} \\
\%p = \left\{ \begin{array}{ll} sm & = \{u : 1/2\} \\ n & = \{u \to u : 1/2, u0 \to u : 1/2\} \\ x & = \{u0 : 1\} \\ r[n,x] & = \{u : 1, u0 : 1\} \end{array} \right\}
\end{array}
$$

Fig. 5. A TVS structure describing a singly linked list pointed to by `x`(cf. Fig. 4).

3.1 TVP

There are two challenging aspects to writing a good TVP specification: one is the design of the instrumentation predicates, which is important for the precision of the analysis; the other is writing the operational semantics manipulating these predicates.

An important observation is that the TVP specification should always be thought of in the terms of the concrete 2-valued world rather than the abstract 3-valued world: the Embedding Theorem guarantees the soundness of the reinterpretation of the formulae in the abstract world. This is an application of the well-known credo of Patrick and Radhia Cousot that the design of a static analysis always starts with a concrete operational semantics.

```
/* Declarations */
%s PVar {x, y, t} // The set of program variables
#include "pred.tvp" // Core and Instrumentation Predicates
%%
/* An Operational Semantics */
#include "cond.tvp" // Operational Semantics of Conditions
#include "stat.tvp" // Operational Semantics of Statements
%%
/* The program's CFG and the effect of its edges */
n1 Set_Null_L(y) n2          // y = NULL;
n2 Is_Null_Var(x) exit       // x == NULL
n2 Is_Not_Null_Var(x) n3     // x != NULL
n3 Copy_Var_L(t, y) n4       // t = y;
n4 Copy_Var_L(y, x) n5       // y = x;
n5 Get_Next_L(x, x) n6       // x = x->n;
n6 Set_Next_Null_L(y) n7     // y->n = NULL;
n7 Set_Next_L(y, t) n8       // y->n = t;
n8 Set_Null_L(t) n2          // t = NULL;
```

Fig. 6. The TVP file for the running example shown in Fig. 1. Files pred.tvp, cond.tvp, and stat.tvp are given in Figures 7, 11, and 12 respectively.

The TVP file is divided into sections separated by **%%**, given in the order described below.

Declarations The first section of the TVP file contains all the declarations needed for the analysis.

Sets. The first declaration in the TVP file is the set *PVar*, which specifies the variables used in the program (here `x`, `y`, and `t`). In the remainder of the specification, set notation allows the user to define the operational semantics for all programs manipulating a certain data type, i.e., it is parametric in *PVar*.

Predicates. The predicates for manipulating singly linked lists as declared in Fig. 1(a) are given in Fig. 7. The **foreach** clause iterates over all the program variables in the set *PVar* and for each of them defines the appropriate core predicate — the unary predicates x, y, and t (**box** tells TVLA to display the predicate as a box). The binary predicate n represents the pointer field n.

For readability, we use some mathematical symbols here that are written in C-like syntax in the actual TVP file (see [10, Appendix B]).

The second **foreach** clause (in Fig. 7) uses *PVar* to define the reachability instrumentation predicates for each of the variables of the program (as opposed to Table 3, which is program specific). Thus, to analyze other programs that manipulate singly linked lists the only declaration that is changed is that of *PVar*.

The fact that the TVP file is specific for the data type L declared in Fig. 1(a) allows us to explicitly refer to n.

```
/* pred.tvp */
foreach (z in PVar) {
    %p z(v₁) unique box // Core predicates corresponding to program variables
}
%p n(v₁, v₂) function // n-field core predicate
%i is[n](v) = ∃v₁, v₂ : (n(v₁, v) ∧ n(v₂, v) ∧ v₁ ≠ v₂) // Is shared instrumentation
foreach (z in PVar) {
    %i r[n, z](v) = ∃v₁ : (z(v₁) ∧ n*(v₁, v)) // Reachability instrumentation
}
%i c[n](v) = ∃v! : n(v, v₁) ∧ n*(v₁, v) // Cyclicity instrumentation
```

Fig. 7. The TVP predicate declarations for manipulating linked lists as declared in Fig. 1 (a). The core predicates are taken from Table 2. Instrumentation predicates are taken from Table 3.

Functional properties. TVLA also supports a concept of *functional properties* borrowed from the database community. Since program variables can point to at most one heap cell at a time, they are declared as **unique**. The binary predicate n represents the pointer field n; the n-field of each list element can only point to at most one target list element, and thus n is declared as a (partial) **function**.

Actions In the second section of the TVP file, we define *actions* that specify the operational semantics of program statements and conditions. An action defines a 2-valued structure transformer. The actions are associated with CFG edges in the third section of the TVP file.

An action specification consists of several parts, each of which is optional (the meaning of these constructs is explained in Sect. 3.2). There are three major parts to the action: (i) Focus formulae (explained in Sect. 3.2), (ii) precondition formula specifying when the action is evaluated, and (iii) update formulae specifying the actual structure transformer. For example, the action `Is_Null_Var(x1)` (see Fig. 11) specifies when the true branch of the condition `x1 == NULL`, is enabled by means of the formula $\neg\exists v : x1(v)$, which holds if `x1` does not point to any list element. Since this condition has no side effects there are no update formulae associated with this action and thus the structure remains unchanged. As another example, the action `Copy_Var_L(x1, x2)` (see Fig. 12) specifies the semantics the statement `x1 = x2`. It has no precondition, and its side effect is to set the $x1$ predicate to $x2$ and the $r[n, x1]$ predicate to $r[n, x2]$.

CFG The third section of the TVP specification is the CFG with actions associated with each of its edges. The edges are specified as **source action target**. The first CFG node that appears in the specification is the entry node of the CFG. The CFG specification for the running example, is given in Fig. 6.

3.2 Process

This section presents a more detailed explanation, using the example shown in Fig. 8, of how the effect of an action associated with a CFG edge is computed. To complete the picture, an iterative (fixed-point) algorithm to compute the result of static-analysis is presented in Sect. 3.3.

Focus First, the Focus operation converts the input structure into a more refined set of structures that represents the same 2-valued structures as the input structure. Given a formula, Focus guarantees that the formula never evaluates to 1/2 in the focused structures. Focus (and Coerce) are semantic reductions (see [5]), i.e., they transfer a 3-valued structure into a set of 3-valued structures representing the same memory states. An algorithm for Focus of a general formula is given in [10]. In the running example, the most interesting focus formula is $\exists v_1 : x(v_1) \wedge n(v_1, v)$, which determines the value of the variable `x` after the `Get_Next_L(x, x)` action (which corresponds to the statement `x = x->n`). Focusing on this formula ensures that $x^S(u)$ is definite at every node u in every structure S after the action. Fig. 8 shows how the structure S_{in} is focused for this action. Three cases are considered in refining S_{in}: (i) The `n`-field of u_0 does not point to any of the list elements represented by u (S_{f0}); (ii) The `n`-field of u_0 points to all of the list elements represented by u (S_{f1}); and (iii) The `n`-field of u_0 points to only some of the list elements represented by u (S_{f2}): u is bifurcated into two nodes — nodes pointed to by the `n`-field of u_0 are represented by $u.1$, and nodes not pointed to by the `n`-field of u_0 are represented by $u.0$.

As explained later, the result can be improved (e.g., S_{f0} can be discarded since u is not reachable from `x`, and yet $r[n, x]^{S_{f0}}(u) = 1$). This is solved by the Coerce operation, which is applied after the abstract interpretation of the statement (see Sect. 3.2).

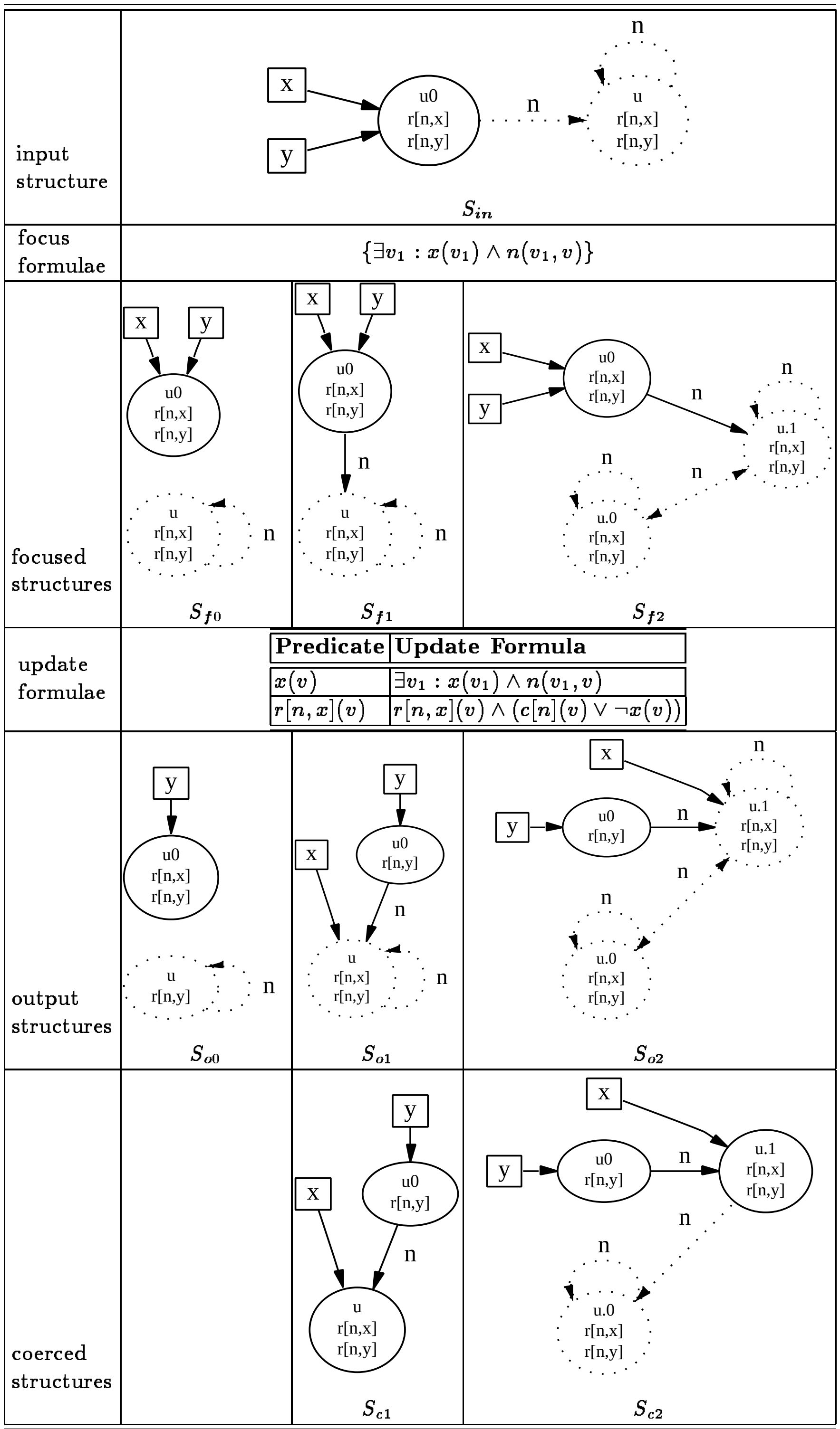

Fig. 8. The first application of abstract interpretation for the statement `x = x->n` in the reverse function shown in Fig. 1.

Preconditions After Focus, preconditions are evaluated. If the precondition formula is potentially satisfied, then the action is performed; otherwise, the action is ignored. This mechanism comes in handy for (partially) interpreting program conditions.

In the running example, the loop `while (x != NULL)` has two outgoing edges in the CFG: one with the precondition $\neg(\exists v : x(v))$, specifying that if x is NULL the statement following the loop is executed (the exit in our case). The other edge has the precondition $\exists v : x(v)$, specifying that if x is not NULL the loop body is executed.

Update Formulae The effect of the operational semantics of a statement is described by a set of update formulae defining the value of each predicate after the statement's action. The Embedding Theorem enables us to reevaluate the formulae on the abstract structures and know that the result provides a conservative abstract semantics. If no update formula is specified for a predicate, it is left unchanged by the action.

In Fig. 8, the effect of the `Get_Next_L` action (`x = x->n`) is computed using the following update formulae: (i) $x(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$, (ii) $r[n,x](v) = r[n,x](v) \wedge (c[n](v) \vee \neg x(v))$. The first formula updates the x variable to be the n-successor of the original x. The second formula updates the information about which nodes are reachable from x after the action: A node is reachable from x after the action if it is reachable from x before the action, except for the node directly pointed to by x (unless x appears on an n-cycle, in which case the node pointed to by x is still reachable even though we advanced to its n-successor). For S_{f2}, the update formula for x evaluates to 1 for $v \mapsto u.1$ and to 0 for all nodes other than $u.1$. Therefore, after the action, the resulting structure S_{o2} has $x^{S_{o2}}(u.1) = 1$ but $x^{S_{o2}}(u.0) = 0$ and $x^{S_{o2}}(u_0) = 0$.

Coerce The last stage of the computation is the Coerce operation, which uses a set of consistency rules (defined in [16, 17, 10]) to make structures more precise by removing unnecessary indefinite values and discarding infeasible structures. The set of consistency rules used is independent of the current action being performed. See [10] for a detailed description of the Coerce algorithm used in TVLA and how TVLA automatically generated consistency rules from the instrumentation predicates and the functional properties of predicates.

For example, Fig. 8 shows how the Coerce operation improves precision. The structure S_{o0} is infeasible because the node u must be reachable from y (since $r[n,y]^{S_{o0}}(u) = 1$) and this is not the case in S_{o0}. In the structure S_{o1}, u is no longer a summary node because x is **unique**; u's self-loop is removed because u already has an incoming n-field and it does not represent a shared list element ($is[n]^{S_{o1}}(u) = 0$). For the same reason, in S_{o2}, $u.1$ is no longer a summary node; Also, the list element represented by $u.1$ already has an incoming n-field and it is not shared ($is[n]^{S_{o2}}(u.1) = 0$), and thus $u.1$'s self-loop is removed. For a similar reason, the indefinite n-edge from $u.0$ to $u.1$ is removed.

Blur To guarantee that the analysis terminates on programs containing loops, we require the number of potential structures for a given program to be finite.

Toward this end, we define the concept of a *bounded structure*. For each analysis, we choose a set of unary predicates called the *abstraction predicates.*[2] In the bounded structure, two nodes u_1, u_2 are merged if $p^S(u_1) = p^S(u_2)$ for each abstraction predicate p. When nodes are merged, the predicate values for their non-abstraction predicates are joined (i.e., the result is 1/2 if their values are different). This is a form of widening (see [5]). The operation of computing this kind of bounded structure is called *Blur*. The choice of abstraction predicates is very important for the balance between space and precision. TVLA allows the user to select the abstraction predicates. By default, all the unary predicates are abstraction predicates, as in the running example.

Example 31 In Fig. 4, the nodes u_0 and u are differentiated by the fact that $x^{S_4}(u_0) = 1$, whereas $x^{S_4}(u) = 0$. (All other predicates are 0.) If x was not an abstraction predicate, then the appropriate bounded structure S'_4 would have had a single node, say u, with $x^{S'_4}(u) = 1/2$ and $n^{S'_4}(u, u) = 1/2$.

After the action is computed and Coerce applied, the Blur operation is used to transform the output structures into bounded structures, thereby generating more compact, but potentially less precise structures.

3.3 Output

Now that we have a method for computing the effect of a single action, what remains is to compute the effect of the whole program, i.e., to compute what structures can arise at each CFG node if the program was used on the given input structures. We use a standard iterative algorithm (e.g., see [12]) with a set of bounded structures as the abstract elements. A new structure is added to the set if the set does not already contain a member that is isomorphic to the new structure. In the running example, the analysis terminates when the structures created in the fourth iteration are isomorphic to the ones created in the third iteration (see Fig. 9). We can see that the analysis precisely captures the behavior of the reverse program.

3.4 Additional Features

One of the main features of TVLA is the support of single structure analysis. Sometimes when the number of structures that arise at each program point is too large, it is better to merge these structures into a single structure that represents at least the same set of 2-valued structures. TVLA enhances this feature even more by allowing the user to specify that some chosen constant number of structures will be associated with each program point.

More specifically, nullary predicates (i.e., predicates of 0-arity) are used to discriminate between different structures. For example, for linked lists we use the

[2] In [16, 17] the abstraction predicates are all the unary predicates.

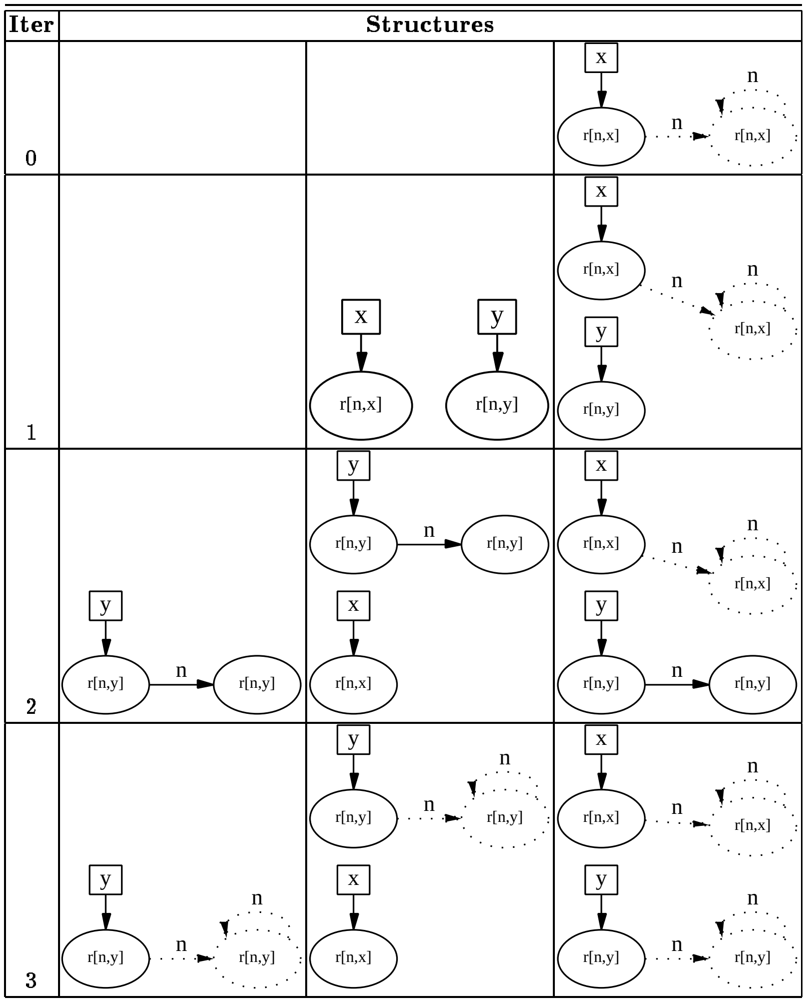

Fig. 9. The structures arising in the reverse function shown in Fig. 1 at CFG node n_2 for the input structure shown in Fig. 4.

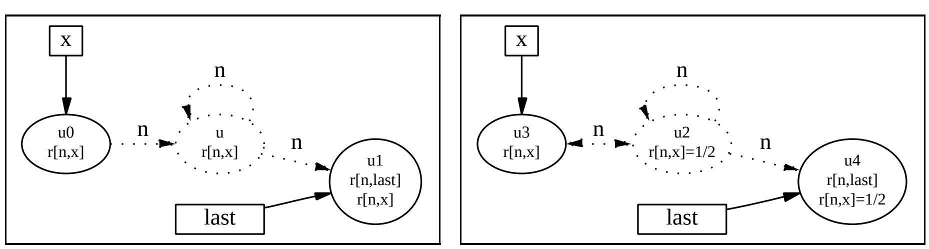

Fig. 10. The structure before and after the rotate function.

predicate $nn[x]() = \exists v : x(v)$ which discriminates between structures in which x actually points to a list element from structures in which it does not. For example, consider a structure S_1 in which both x and y point to list elements, and another structure S_2 in which both x and y are NULL. Merging S_1 and S_2 will loose the information that x and y are simultaneously allocated or not allocated. Notice that S_1 has $nn[x] = nn[y] = 1$ and S_2 has $nn[x] = nn[y] = 0$ therefore S_1 and S_2 will not be merged together.

In some cases (such as safety analysis of Mobile Ambients, see [13]) this option makes an otherwise infeasible analysis run in a reasonable time. However, there are other cases in which the single-structure method is less precise or even more time consuming than the usual method, which uses sets of structures.

TVLA also supports modeling statements that handle dynamically allocated and freed memory.

4 A Case Study - Singly Linked Lists

We used the functions analyzed in [7] with sharing and reachability instrumentation predicates (see Appendix A). The same specification for the operational semantics of pointer-manipulating statements was used for each of the functions was written once and used with each of the CFGs.

Most of the analyses were very precise, and running times were no more than 8 seconds for even the most complex function (merge).

The `rotate` function performs a cyclic shift on a linked-list. The analysis of this example is not as precise as possible (see Fig. 10). The indefinite edge from u_2 to u_3 is superfluous and all the list elements should be known to be reachable from x. The imprecision arises because the list becomes cyclic in the process, and the 3-valued evaluation of the reachability update-formula in the action `Set_Next_Null_L` (see Fig. 12) is not very precise in the case of cyclic lists. A simple rewriting of the program to avoid the temporary introduction of a cycle state would have solved the problem.

The `merge` function, which merges two ordered linked-lists, is a good example of how extra instrumentation (reachability) can improve the space consumption of the analysis. Analyzing the merge function without the reachability predicate creates tens of thousands of graphs and takes too much space. Adding the reachability predicate as an instrumentation reduces the number of graphs to 327 and the time to about 8 seconds.

5 Conclusion

The method of 3-valued-logic-based static analysis can handle a wider class of problems than shape analysis. We have successfully analyzed Mobile Ambients [13] even though it is a completely different sort of language for specifying computation. We can also show partial correctness of algorithms such as sorting programs [11].

However, it is clear that some analyses go beyond the scope of TVLA, and it is not obvious whether TVLA can or should be extended to support them. Specifically, the operational semantics must be expressible using first-order logic with transitive closure; in particular, no explicit arithmetic is currently supported, although it can be defined using predicate symbols. Also, the set of predicate symbols is fixed.

The system was implemented in Java, which is an Object-Oriented imperative language. The use of libraries, such as the Collections library, enabled us to incorporate fairly complex data structures without using a more high-level language, such as ML.

Static-analysis algorithms are hard to design, prove correct, and implement. The concept of 3-valued-logic-based analysis greatly simplifies the problem, because it allows us to work with the concrete operational semantics instead of the abstract semantics. The use of 3-valued logic guarantees that the transition to the abstract semantics is sound. TVLA introduces two major contributions toward the simplification of the problem. First, it provides a platform on which one can easily try new algorithms and observe the results. Second, it contains system and algorithmic support for instrumentation information.

5.1 The Essence of Instrumentation

Our experience indicates that instrumentation predicates are essential to achieving efficient and useful analyses. First, they are helpful in debugging the operational semantics. The instrumentation predicates are updated separately from the core predicates, and any discrepancy between them is reported by the system. Our experience indicates that in many cases this reveals bugs in the operational semantics.

The conventional wisdom in static analysis is that there is a trade-off between the time of analysis and its precision (i.e., that a more precise analysis is more expensive). In case of 3-valued-logic-based analysis, this is not always true. Often it happens that an analysis that uses more instrumentation predicates creates fewer unneeded structures, and thus runs faster. A good example of this is the merge function (see Sect. 4) where adding the reachability information drastically reduces both the space and the time needed for the analysis.

In general, the introduction of instrumentation predicates is a very good tool for improving precision, and has a very low cost. If a property holds for many but not all nodes of the structures that arise in a program, then we can use an instrumentation predicate to track at which program points and for which nodes the property holds. This allows us to use the implications of the property without limiting ourselves to programs where the property holds . For example, we use cyclicity instrumentation to update the reachability information. If a singly linked list is acyclic, updating the reachability information can be done more precisely. The use of cyclicity instrumentation allows us to take advantage of this property without limiting the analysis to programs in which lists are always acyclic. Of course, in some programs, such as `rotate`, where cyclicity is temporarily introduced, we may lose precision when evaluating formulae in 3-valued logic. This is in line with the inherent complexity of these problems. For example, updating reachability in general directed graphs is a difficult problem.

Formally, instrumentation predicates allow us to narrow the set of 2-valued structures represented by a 3-valued structure, and thereby avoid making overly conservative assumptions in the abstract interpretation of a statement. For example, the structure shown in Fig. 4 represents an acyclic singly linked list, which means that all of the list elements represented by u are not shared. Thus, $is[n]^{S_4}(u) = 0$. The same holds for u_0. Without the sharing information, the structure might also represent 2-valued structures in which the linked list ends with a cycle back to itself.

For unary instrumentation predicates, we can fine-tune the precision of an analysis by varying the collection of predicates used as abstraction predicates. The more abstraction predicates used, the finer the distinctions that are made, which leads to a more precise analysis. For example, the fact that *is* is an abstraction predicate allow us to distinguish between shared and unshared list elements in programs such as the `swap` function, where a list element is temporarily shared. Of course, this may also increase the cost of the analysis.

5.2 Theoretical Contributions

Space precludes us from a comprehensive comparison with [17]. The Coerce algorithm was optimized to avoid unnecessary recomputations by using lazy evaluation, imposing an order of constraint evaluation and using relational database query optimization techniques (see [19]) to evaluate formulae. The Focus algorithm was generalized to handle an arbitrary formula. This was crucial to support the formulae used for analyzing sorting programs. In addition, the Focus algorithm in TVLA was also optimized to take advantage of functional properties of the predicates.

The worst-case space of the analysis was improved from doubly exponential to singly exponential by means of the option in which all the structures with the same nullary predicate values are merged together. Thus, the number of potential structures becomes independent of the number of nodes in the structure. Interestingly, in most of the cases analyzed to date the analysis remains rather precise. However, in some cases it actually increases the space needed for the analysis due to decreased precision.

5.3 Other Analysis Engines

The main advantages of TVLA over existing systems, such as [1, 18, 2], are: (i) quick prototyping of non-trivial analyses (e.g., sorting); (ii) good control over precision through instrumentation predicates; (iii) good separate control over space requirements through abstraction predicates and the single-structure option; and (iv) the abstract semantics is automatically derived from the concrete operational semantics (i.e., there is no need to specify the abstract semantics directly, which is quite complicated for shape-analysis). However, TVLA currently is intra-procedural only, there is an ongoing research to extend TVLA to handle recursive procedures (see [14]).

5.4 Further Work

The system is very useful in the analysis of small programs. However, there are many theoretical and implementation issues that need to be solved before the analysis can scale to larger programs.

An operational semantics of the instrumentation predicates needs to be specified for all programming language constructs, which can be error-prone. In the future, it may be possible to generate such update formulae for a subset of first-order logic.

The choice of the abstraction predicates is very important for the space/precision trade-off. We lack a good methodology for selecting these predicates.

The major problem in terms of scalability of the system is the space needed for the analysis. We have devised some techniques to alleviate the problem, but they are not enough. A possible solution to the problem may be to use Binary Decision Diagrams (BDDs) to represent logical structures ([3]). Another possible solution is the use of secondary storage.

Acknowledgements

We are grateful for the helpful comments and contributions of N. Dor, M. Fähndrich, G. Laden, F. Nielson, H.R. Nielson, T. Reps, N. Rinetskey, R. Shaham, O. Shmueli, R. Wilhelm, and A. Yehudai.

References

1. M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *SAS'95, Static Analysis Symposium*, LNCS 983, pages 33–50. Springer, September 1995.
2. U. Aßmann. *Graph Grammar Handbook*, chapter OPTIMIX, A Tool for Rewriting and Optimizing Programs. Chapman-Hall, 1998.
3. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *Computing Surveys*, 24(3):293–318, September 1992.
4. 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.
5. 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.
6. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.
7. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *SAS'00, Static Analysis Symposium*, 2000.
8. D. Evans. Static detection of dynamic memory errors. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 1996.

9. 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.
10. 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.
11. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, 2000. Available at http://www.cs.wisc.edu/~reps.
12. S.S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan & Kaufmann, third edition, 1999.
13. F. Nielson, H.R. Nielson, and M. Sagiv. A kleene analysis of mobile ambients. In *Proceedings of the 2000 European Symposium On Programming*, March 2000.
14. N. Rinetskey and M. Sagiv. Interprocedual shape analysis for recursive programs. Available at http://www.cs.technion.ac.il/~maon, 2000.
15. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, January 1998.
16. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999.
17. 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, March 2000. Submitted for publication. Available at "http://www.cs.wisc.edu/wpis/papers/tr1383.ps".
18. S.W.K. Tjiang and J. Hennessy. Sharlit—a tool for building optimizers. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 82–93, June 1992.
19. J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies.* Comp. Sci. Press, Rockville, MD, 1989.

A Empirical Results

The system was used to analyze on a number of examples (see Sect. 4). The timing information for all the functions analyzed is given in Table 4.

B A TVP File for Shape Analysis on Programs Manipulating Singly Linked Lists

The actions for handling program conditions that consists of pointer equalities and inequalities are given in Fig. 11.

The actions for manipulating the `struct node` declaration from Fig. 1(a) are given in Fig. 12. The actions `Set_Next_Null_L` and `Set_Next_L` model destructive updating (i.e., assignment to `x1->n`), and therefore have a nontrivial specification.

We use the notation $\varphi_1 ? \varphi_2 : \varphi_3$ for an if-then-else clause. If φ_1 is 1 then the result is φ_2, if φ_2 is 0 then the result is φ_3. If φ_1 is 1/2 then the result is $\varphi_2 \sqcup \varphi_3$. We use the notation $TC(v_1, v_2)(v_3, v_4)$ for the transitive-closure operator. The variables v_3 and v_4 are the free variables of the sub-formula over which the transitive closure is performed, and v_1 and v_2 are the variables used on the resulting binary relation.

Table 4. Description of the singly-linked-list programs analyzed and their timing information. These programs are collections of interesting programs from LCLint [8], [9], Thomas Ball, and from first-year students. They are available at `http://www.math.tau.ac.il/~nurr`.

Program	Description	Time (seconds)	Number of Structures
search	searches for an element in a linked list	40	0.708
null_deref	searches a linked list, but with a typical error of not checking for the end of the list	48	0.752
delete	deletes a given element from a linked list	145	2.739
del_all	deletes an entire linked list	11	0.42
insert	inserts an element into a sorted linked list	140	2.862
create	prepends a varying number of new elements to a linked list	21	0.511
merge	merges two sorted linked lists into one sorted list	327	8.253
reverse	reverses a linked list via destructive updates	70	1.217
fumble	an erroneous version of reverse that loses the list	81	1.406
rotate	performs a cyclic rotation when given pointers to the first and last elements	25	0.629
swap	swaps the first and second elements of a list, fails when the list is 1 element long	31	0.7
getlast	returns the last element of the list	40	0.785
insert_sort	sorts a linked list using insertion sort	3773	160.132
bubble_sort	sorts a linked list using bubble sort	3946	186.609

```
/* cond.tvp */
%action Is_Not_Null_Var(x1) { %t x1 + " != NULL"
    %f { x1(v) } %p ∃v : x1(v)
}
%action Is_Null_Var(x1) { %t x1 + " == NULL"
    %f { x1(v) } %p ¬(∃v : x1(v))
}
%action Is_Eq_Var(x1, x2) { %t x1 + " == " + x2
    %f { x1(v), x2(v) }
    %p ∀v : x1(v) ⇔ x2(v)
}
%action Is_Not_Eq_Var(x1, x2) { %t x1 + " != " + x2
    %f { x1(v), x2(v) }
    %p ¬∀v : x1(v) ⇔ x2(v)
}
```

Fig. 11. An operational semantics in TVP for handling pointer conditions.

```
/* stat.tvp */
%action Set_Null_L(x1) { %t x1 + " = NULL"
    { x1(v) = 0    r[n,x1](v) = 0}
}
%action Copy_Var_L(x1, x2) { %t x1 + " = " + x2
    %f { x2(v) }
    { x1(v) = x2(v)    r[n,x1](v) = r[n,x2](v)}
}
%action Malloc_L(x1) { %t x1 + " = (L) malloc(sizeof(struct node))) "
    %new
    { x1(v) = isNew(v)    r[n,x1](v) = isNew(v) }
}
%action Free_L(x1) { %t "free(x1)"
    %f {x1(v)}
    %message ∃v_1,v_2 : x1(v_1) ∧ n(v_1,v_2) ->
                "Internal error! assume that " + x1 + "->" + n + "==NULL"
    %retain ¬x1(v)
}
%action Get_Next_L(x1, x2) { %t x1 + " = " + x2 + "->" + n
    %f { ∃v_1 : x2(v_1) ∧ n(v_1,v)}
    { x1(v) = ∃v_1 : x2(v_1) ∧ n(v_1,v)
      r[n,x1](v) = r[n,x2](v) ∧ (c[n](v) ∨ ¬x2(v))}
}
%action Set_Next_Null_L(x1) { %t x1 + "->" + n + " = NULL"
    %f { x1(v) }
    { n(v_1,v_2) = n(v_1,v_2) ∧ ¬x1(v_1)
      is[n](v) = is[n](v) ∧ (¬(∃v_1 : x1(v_1) ∧ n(v_1,v))∨
                    ∃v_1,v_2 : (n(v_1,v) ∧ ¬x1(v_1)) ∧ (n(v_2,v) ∧ ¬x1(v_2)) ∧ v_1 ≠ v_2)
      r[n,x1](v) = x1(v)
      foreach(z in PVar-{x1}) {
          r[n,z](v) =(c[n](v) ∧ r[n,x1](v)?
                    z(v) ∨ ∃v_1 : z(v_1) ∧ TC(v_1,v)(v_3,v_4)(n(v_3,v_4) ∧ ¬x1(v_3)) :
                    r[n,z](v) ∧ ¬(r[n,x1](v) ∧ ¬x1(v) ∧ ∃v_1 : r[n,z](v_1) ∧ x1(v_1)))
      }
      c[n](v) = c[n](v) ∧ ¬(∃v_1 : x1(v_1) ∧ c[n](v_1) ∧ r[n,x1](v))}
}
%action Set_Next_L(x1, x2) { %t x1 + "->" + n + " = " + x2
    %f { x1(v), x2(v) }
    %message ∃v_1,v_2 : x1(v_1) ∧ n(v_1,v_2) ->
                "Internal error! assume that " + x1 + "->" + n + "==NULL"
    { n(v_1,v_2) = n(v_1,v_2) ∨ x1(v_1) ∧ x2(v_2)
      is[n](v) = is[n](v) ∨ ∃v_1 : x2(v) ∧ n(v_1,v)
      foreach(z in PVar) {
          r[n,z](v) = r[n,z](v) ∨ r[n,x2](v) ∧ ∃v_1 : r[n,z](v_1) ∧ x1(v_1)
      }
      c[n](v) = c[n](v) ∨ (r[n,x2](v) ∧ ∃v_1 : x1(v_1) ∧ r[n,x2](v_1))}
}
```

Fig. 12. An operational semantics in TVP for handling the pointer-manipulation statements of linked lists as declared in Fig. 1(a).