Data Structures and Algorithms for Efficient Shape Analysis

Roman Manevich[1]
School of Computer Science, Tel-Aviv University, Israel

January, 2003

[1]rumster@tau.ac.il

Acknowledgements

I would like to express my sincere gratitude to all those who contributed to the completion of my thesis.

First and foremost, I would like to express my deep gratitude, respect, and thanks to Dr. Shmuel (Mooly) Sagiv, my supervisor. This thesis would have been impossible to complete without his constant encouragement, guidance, patience and support.

A warm thanks to IBM's Advanced Programming Tools group at the T.J. Watson Research center where I visited during the summer of 2001. Part of this work was done in close cooperation with John Field, Ganesan Ramalingam and Deepak Goyal.

To Prof. Thomas Reps, and Dr. Haim Kaplan, for valuable and interesting discussions.

To my colleagues from the Tel-Aviv Programming Languages group: Eran Yahav, Nurit Dor, Ran Shaham, Greta Yorsh, Noam Rinetzky, Alex Warshavski and Michael Pan.

To Ronen Gvili, Nir Andelman, Yossi Mossel and several other colleagues from the school of computer sciences, for backing me and cheering me up.

To the Israeli Academy of Science, for their generous financial support.

To Ramot and the G&D research program, for their financial support.

Abstract

Shape analysis concerns the problem of determining "shape invariants" for programs that perform destructive updating on dynamically allocated storage. TVLA (**T**hree-**V**alued **L**ogic **A**nalysis) [LAS00] is a system for shape analysis that can be used in different ways to create different shape analysis implementations that provide varying degrees of efficiency and precision.

A key property of TVLA is that the stores that can possibly arise during execution are represented (conservatively) using sets of 3-valued logical structures.

While the TVLA system is general, this generality comes at a very high price:

"Often the analysis of small programs is very expensive in terms of time and space."

Indeed, the algorithms used in TVLA have very high complexity. The overall complexity could be doubly exponential in the program size.

This thesis is aimed at improving the performance of static analyses in the TVLA system, with the aid of new algorithms and data structures. In the first part, new data structures, which represent 3-valued first-order structures are shown to largely reduce the space consumption. One of these data structures is based on OBDDs (*ordered binary decision diagrams*) which are widely used for finite state model checking. In the second part, a reduction in the overall analysis cost is achieved by exploring new abstractions that produce more compact sets of structures.

Contents

Chapter 1

Introduction

1.1 Motivation

Shape analysis concerns the problem of determining "shape invariants" for programs that perform destructive updating on dynamically allocated storage.

TVLA (**T**hree-**V**alued **L**ogic **A**nalysis) [LAS00] is a system for shape analysis that can be used in different ways to create different shape analysis implementations that provide varying degrees of efficiency and precision. TVLA is designed to model dynamic allocation precisely by representing program states as sets of *first-order structures*. A first-order representation uses a finite collection of *predicates* to define states; the predicates range over a universe of *individuals* that may evolve—expand and contract—during analysis. The use of first-order structures permits, e.g., dynamic memory allocation or dynamic thread creation to be modeled in a natural way [SRW02, Yah01]. TVLA has been successfully applied to a wide variety of deep program analysis and verification problems, including analyzing C programs for "cleanness" (statically checking for memory leaks and dangling or uninitialized pointers), verifying the correctness of a simple garbage collector, determining whether clients of a Java library satisfy the library's *conformance constraints* for correct usage [RWF$^+$02], and verifying certain safety properties of a packet router encoded in the Mobile Ambient calculus [CG98]. The appeal of TVLA can be contributed to the following factors:

- **Expressiveness** The language in which users express their shape analysis algorithms is first-order predicate logic with transitive closure. This is a very expressive language, allowing users to express many useful properties that the algorithms seek to predict.

- **Soundness for Free** The problem of discovering shape invariants is not decidable. A common method for dealing with undecidability is via abstraction [CC79], which makes it possible to give approximate solutions. Proving the soundness of static analysis algorithms is usually a very time-consuming and complex task. TVLA is based on a firm theoretical background [SRW02], which lifts this burden. TVLA users can quickly generate and experiment with quite complex algorithms without having to implement the abstraction and without having to worry about the soundness of the generated algorithm.

- **Flexibility** TVLA employs powerful techniques for refining and controlling the precision (level of approximation) of its analyses, which makes it possible to achieve very accurate verification results.

However, the same factors that make TVLA so appealing also make it very expensive in terms of space and time: The worst-case space complexity of the default algorithm is doubly-exponential in the size of the program for some analyses. Although TVLA analyses do not always exhibit the worst case behavior, making it possible to solve small programs, scalability is a major issue. As indicated in [LAS00], a main problem is the space needed for the analysis.

TVLA employs an iterative fix-point algorithm. The number of iterations that are required to arrive to the solution can be very high, making the analysis very slow even for small programs (several hours in some cases and in other cases not terminating even after several days). The number of iterations and amount of overall work of the algorithm is directly related to the amount of structures it produces.

In this thesis, we attempt to attack the scalability issue. We reduce the space consumption of the analysis by designing new data structures that represent logical structures by sharing common information. We then proceed to reduce the overall work of the algorithm by reducing the number of structures it produces. This is achieved by means of abstractions, which allow us to represent more compactly large amounts of information by conservative approximations.

1.2 Main Results

This research reports both theoretical and empirical results for tackling the scalability problem of shape analysis:

1. We describe the properties required from the data structures that represent logical structures and then describe and evaluate two structure representation techniques. One uses OBDDs; the other uses a variant of a *functional map* data structure. Using a suite of benchmark analysis problems, we systematically compare the new representations with TVLA's existing state representation. The results show that both the OBDD and functional implementations reduce space consumption in TVLA by a factor of 4 to 10 relative to the current TVLA state representation, without compromising analysis time.

2. We describe new abstraction techniques and compare them with the existing abstractions on a suite of benchmarks. The results show that the new abstraction techniques maintain the precision achieved by the most precise existing technique. At the same time, the new

```
/* 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)

Figure 1.1: (a) Declaration of a linked-list data type in C. (b) A C function that uses destructive updating to reverse the list pointed to by parameter `x`.

abstractions substantially reduce the amount of work required to compute the solution, sometimes by several orders of magnitude.

1.3 Outline of the Thesis

The thesis is organized as follows: Chapter 2 contains a short primer on TVLA that describes the parts relevant for this thesis and a short introduction to OBDDs; Chapter 3 describes the new representation techniques for logical structures; Chapter 4 describes the new abstraction techniques; and Chapter 5 concludes the thesis.

A program that destructively reverses a singly linked list is shown in Fig. 1.1. The shape analysis of this program serves as a running example in this thesis.

Chapter 2

Background

2.1 TVLA Primer

In this section, we briefly repeat those aspects of TVLA that will be used in the sequel of this thesis. Complete details of the system may be found in [LAS00].

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 2.1. We say that the values 0 and 1 are *definite values* and that $1/2$ is an *indefinite value*, and define a partial order $\sqsubseteq$ on truth values to reflect information content: $l_1 \sqsubseteq l_2$ denotes that l_1 has more definite information than l_2:

Definition 2.1.1 [Information Order]. *For $l_1, l_2 \in \{0, 1/2, 1\}$, we define the* **information order** *on truth values as follows: $l_1 \sqsubseteq l_2$ if $l_1 = l_2$ or $l_2 = 1/2$. The symbol $\sqcup$ (join) denotes the least-upper-bound operation with respect to $\sqsubseteq$, i.e., $l_1 \sqcup l_2 = l_1$, if $l_1 = l_2$ and $1/2$ otherwise.*

Kleene's semantics of 3-valued logic is monotonic in the information order.

2.1.1 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

$\wedge$	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 2.1: Kleene's 3-valued interpretation of the propositional operators.

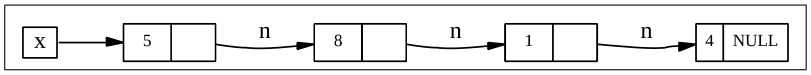


Figure 2.1: A possible store for the running example.

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.

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.)

In TVLA, predicates are required to have arity ≤ 2; hence it is natural to depict structures in the form of directed graphs. Consider the 2-valued structure S_0 depicted in Fig. 2.2. Here, the variable x is represented by a unary predicate x, which has value 1 only for u_1. In general, a unary predicate p which holds for a node u is drawn inside the node u, not shown if its value is 0. A unary predicate that represents a pointer variable is represented graphically by a solid arrow connecting p to each individual u for which $p(u) = 1$, and no arrow otherwise. If p is 0-valued for all individuals, the predicate name p is not depicted; e.g., in S_0, the absence of predicate y indicates that variable y is null in S_0. A solid directed edge labeled by p from u_1 to u_2 denotes the fact that $p(u_1, u_2) = 1$. The name of a node is written inside the node using an italic face. Node names are only used for ease of presentation and do not affect the analysis.

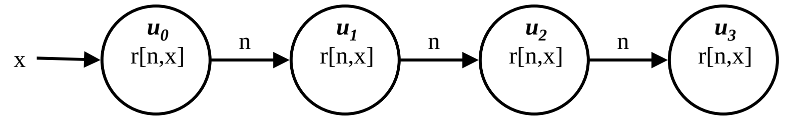
In the running example, a 2-valued structure represents a memory state (also called a *store*); an individual corresponds to a list element. The store in Fig. 2.1 is represented by the 2-valued structure $S_{2.2}$ shown in Fig. 2.2. The structure $S_{2.2}$ has four nodes, u_0, u_1, u_2, and u_3 representing the four list elements. This representation intentionally ignores the values of the data field, which are usually immaterial for the analysis.

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. 2.2, the variable x is represented by the unary predicate x, which is 1 only for 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 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 $S_{2.2}$ in Fig. 2.2 has $r[n, x]^{S_{2.2}}$ set to 1 for all the nodes because they are all reachable from x.

The 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. 2.2, all the elements of the list are unshared, and thus $is[n]^{S_{2.2}}$ is 0 for all of them.

The predicate $c[n]$ holds for nodes on a cycle of accesses along n-fields. In Fig. 2.2, the list is acyclic, and thus $c[n]^{S_{2.2}}$ is 0 for all of the nodes.



	x	y	t	$r[n,x]$	$r[n,y]$	$r[n,t]$	$is[n]$	$c[n]$
u_0	1	0	0	1	0	0	0	0
u_1	0	0	0	1	0	0	0	0
u_2	0	0	0	1	0	0	0	0
u_3	0	0	0	1	0	0	0	0

n	u_0	u_1	u_2	u_3
u_0	0	1	0	0
u_1	0	0	1	0
u_2	0	0	0	1
u_3	0	0	0	0

Figure 2.2: A logical structure $S_{2.2}$ representing the store shown in Fig. 2.1 in a graphical and tabular representation.

2.1.2 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

$$\exists v_1 : (x(v_1) \land n^*(v_1, v)) \tag{2.1}$$

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.

2.1.3 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 $p^S\colon (U^S)^k \to \{0, 1, 1/2\}$, 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. Unary indefinite predicate values are drawn inside the nodes and marked as indefinite (this does not occur in the running example).

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. 2.2, the nodes u_1, u_2, and u_3 are represented by the single node u in Fig. 2.3.

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 double circles. 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 Section 2.2.

Embedding

Although structures may have different individuals, we can define an order on structures, denoted by $\sqsubseteq$ based on the concept of *embedding*. The goal is to guarantee that if $S \sqsubseteq S'$ then the value of every formula in S is less or equal to its value in S'. In particular, whenever the formula evaluates to a definite value in S' then the formula has the same value in S. Formally,

Definition 2.1.2 *Let S and S' be two structures. Let $f\colon U^S \to U^{S'}$ be surjective. We say that f* **embeds** *S in S' (denoted by $S \sqsubseteq^f S'$) if (i) for every predicate p (including sm) of arity k and all $u_1, \ldots, u_k \in U^S$,*

$$p^S(u_1, \ldots, u_k) \sqsubseteq p^{S'}(f(u_1), \ldots, f(u_k)) \tag{2.2}$$

and (ii) for all $u' \in U^{S'}$

$$(|\{u \mid f(u) = u'\}| > 1) \sqsubseteq sm^{S'}(u') \tag{2.3}$$

We say that S **can be embedded in** *S' (denoted by $S \sqsubseteq S'$) if there exists a function f such that $S \sqsubseteq^f S'$.*

A special kind of embedding is a *tight embedding*, in which information loss is minimized when multiple individuals of S are mapped to the same individual in S':

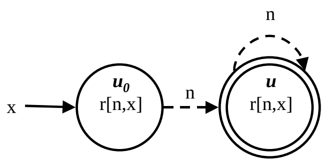
Definition 2.1.3 *A structure S' is a* **tight embedding** *of S if there exists a surjective function $blur : U^S \to U^{S'}$ such that, for every $p \neq sm$ of arity k,*

$$p^{S'}(u'_1, \ldots, u'_k) = \bigsqcup_{blur\ (u_i)=u'_i, 1\leq i\leq k} p^S(u_1, \ldots, u_k) \tag{2.4}$$

and for every $u' \in U^{S'}$,

$$sm^{S'}(u') = (|\{u | blur\ (u) = u'\}| > 1) \sqcup \bigsqcup_{blur\ (u)=u'} sm^S(u) \tag{2.5}$$

Because $blur$ is surjective, equations (2.4) and (2.5) uniquely determine S' (up to isomorphism); therefore, we say that $S' = blur\ (S)$.



	x	y	t	$r[n,x]$	$r[n,y]$	$r[n,t]$	$is[n]$	$c[n]$
u_0	1	0	0	1	0	0	0	0
u	0	0	0	1	0	0	0	0

n	u_0	u
u_0	0	1/2
u	0	1/2

Figure 2.3: A 3-valued structure $S_{2.3}$ representing lists of length 2 or more that are pointed to by program variable x (e.g., $S_{2.2}$).

Example 2.1.4 In the running example, the 3-valued structure $S_{2.3}$ shown in Fig. 2.3 represents the 2-valued structure $S_{2.2}$ for $f(u_0) = u_0$ and $f(u_1) = f(u_2) = f(u_3) = u$. In fact, the structure shown in Fig. 2.3 represents all the singly-linked lists with two or more elements pointed to by x.

The unary predicate symbol x has $x^{S_{2.3}}(u_0) = 1$, indicating that the program variable x is known to point to the list element represented by u_0, and $x^{S_{2.3}}(u) = 0$, indicating that x is known not to point to any of the list elements represented by u.

The binary predicate symbol n has $n^{S_{2.3}}(u_0, u) = 1/2$, 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. 2.2) — but does not point to all the list elements represented by u (e.g. u_2 in Fig. 2.2). Also, $n^{S_{2.3}}(u, u) = 1/2$, 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. 2.2 the n-field of u_2 points to u_3, but not to u_1).

Consider the program shown in Fig. 1.1, which reverses a singly-linked list in place. As an expository example, let us say that we wish to confirm that the program never dereferences a pointer that is NULL. Table 2.2 contains the predicates used for the shape analysis of the running example.

2.1.4 Abstract Interpretation in TVLA

TVLA takes as input a control flow graph (CFG) of the program to be analyzed—each edge of which can be thought of as being annotated with a sequence of *actions*—and an abstract representation of a set of initial states. Each action is an operation on the abstract data type (ADT) used to represent first-order structures (the full set of actions is enumerated in Section 3.1), and the action sequences associated with CFG edges collectively encode those aspects of the program's semantics that are relevant to the program analysis problem at hand. The TVLA user specifies action sequences using a high-level programming language whose core constructs

Table 2.2: The predicates used for shape analysis of the program in Fig. 1.1 and their meaning.

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?
$r[n, x](v)$	Is v reachable from program variable `x`using field `n`?
$r[n, y](v)$	Is v reachable from program variable `y`using field `n`?
$r[n, t](v)$	Is v reachable from program variable `t`using field `n`?
$is[n](v)$	Is v pointed to by more than one `n`-field?
$c[n](v)$	Does v reside on a directed cycle via dereferences along `n`-fields?
$sm(v)$	Can v represent multiple concrete heap cells?

are based on first-order logic extended with transitive closure. During analysis, the high-level TVLA operations are interpreted, and translated into action sequences.

The most important high-level TVLA construct is the *predicate update* operation, which is used to encode a state update. For example, the statement `x = x->n` in Fig. 1.1 is modeled using a predicate update operation of the form $x(v) := \exists v_1 : x(v_1) \wedge n(v_1, v)$. This update operation is translated into a loop containing structure ADT actions whose effect is: to evaluate the right-hand side of the update operation for each possible binding of the free variable v to an individual in the current structure, and to bind the result to v in the interpretation of x in a new copy of the structure.

TVLA carries out abstract interpretation by exhaustively exploring the abstract program states derivable from the initial state set via action sequences associated with CFG edges. The iterative algorithm used to perform an abstract interpretation over the input program is outlined in Fig. 2.4. The actions which may be associated with each CFG edge are enumerated in Fig. B.1, and described in greater detail in Section 3.1. The actions associated with edges are always terminated by actions that carry out a *blur* operation, and therefore only blurred structures are added to the state space. This is explained in more detail in the next section.

The structures generated by abstract interpretation of the first iteration of the loop body of the `reverse` function are depicted in Fig. 2.5. The analysis begins with the 3-valued structure S_0. Then, the actions modeling the statement `y = x` have the effect of setting `y` to point to u_1, resulting in the structure S_1. The most interesting case is the assignment `x = x->n`. This statement is modeled by the predicate update action for `x = x->n` described above, preceded by a *focus* action. The focus action, which we will not describe in detail here, has the effect of bifurcating the incoming structure into a *set* of structures; the basic idea is to replace a structure in which a predicate p has value $1/2$ by a pair of structures, one where p has value 1, and one where p has value 0. Focus allows us to do a precise "case analysis" on the resulting

```
initialize($S_0$) {
    for each $S \in S_0$ {
        push(stack, $[P_{entry}, S]$)
    }
}

explore() {
  while stack is not empty {
    $[P, S]$ = pop(stack)
    if add($stateSpace(P)$, immutableCopy($S$)) {
      for each outgoing CFG edge $(P, P')$ {
        $S'$ = copy($S$)
        Apply (Action($P, P'$), $S'$)
        push(stack, $[P', S']$)
      }
    }
  }
}
```

Figure 2.4: The structure of TVLA's abstract interpretation algorithm. P and P' denote CFG nodes where P_{entry} is the entry node. S and S' denote program states, represented by first-order structures, where S_0 is the set of initial program states.

set of structures, which frequently sharpens the analysis results. Here, for example, focus allows us to distinguish the case where x is null (represented by structure $S_{2.0}$) from the cases where x is non-null; this distinction is critical for verifying that the when the statement x = x->n is executed, it does not cause a null dereferencing to the pointer field n. The *coerce* action, which we will also not describe in detail is then able to determine that the structure S_0 is inconsistent and does not represent a store that can possibly arise from any execution of the program. This is because the node u is found to be reachable from the pointer variable x (it has $r[n, x](u) = 1$) while x is NULL. Coerce infers this inconsistency by solving a set of constraints. The set of constraints, automatically produced for this analysis, contains the constraint $\neg\exists v_1 : x(v_1) \wedge n^*(v_1, v) \implies \neg r[n, x](v)$, which evaluates to 0 for $v = u$. This breach of constraint causes the analysis to discard the structure and avoid emitting a false alarm.

2.2 Abstraction in TVLA

The complexity of static analysis algorithms largely depends on the abstractions used. Abstraction is achieved by choosing an appropriate *abstract domain* to compactly represent the concrete properties of a program. A central problem in static analysis is finding a suitable domain that provides a good balance between efficiency and the level of precision. This is a very tough problem that currently lacks a general theory for expressing the relationship between efficiency and precision mathematically. Although for some cases there are optimal solutions (in the sense of join over all paths), in almost all non-trivial analyses research is limited to an empirical evaluation of different abstract domains. The remainder of this section briefly describes the abstract domains used in TVLA and the related techniques.

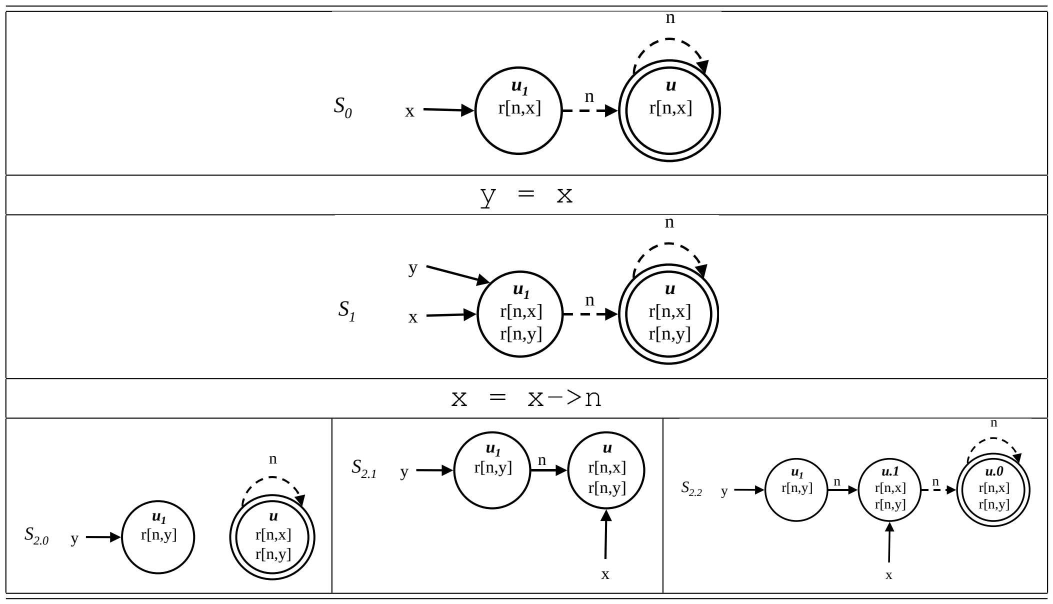


Figure 2.5: Structures generated by abstract interpretation of the first iteration of the loop in the `reverse` function.

In order to ensure termination, TVLA introduces the concept of a *bounded structure*. We separate the set of unary predicates into two groups — the abstraction predicates and the non-abstraction predicates. A bounded structure is obtained from a 3-valued structure by merging individuals with the same set of unary abstraction predicate values using the *blur* operation. The ordered set of unary abstraction predicate values for an individual is called its *canonic name*. This procedure limits the number of individuals for a structure with p unary abstraction predicates to at most 3^p. This alone does not ensure termination, since numerous different (but possibly *isomorphic*) structures can be propagated into the same CFG node when it contains cycles. The additional condition that is needed in order to ensure termination is limiting the size of the set of structures in any CFG node. There are two existing methods to ensure this — the relational method and the single-structure method.

2.2.1 The Relational Method

This method is based on the following observation. Given that the structures are bounded, there can only be a finite number of them that are not *isomorphic* to one another[1]. To see this, notice that two structures can be different, i.e., not isomorphic to one another, if their set of canonic names are different. In addition, two structures with the same set of canonic names can be different if a non-abstraction predicate of arity $k > 1$ has a different interpretation for the two

[1]The notion of structure isomorphism is defined in [LAS00]. Informally, two structures are isomorphic when there is a one-to-one mapping between their individuals that preserves all predicate values.

structures and for a k-tuple of nodes with the same canonic name. These are the only reasons that can cause structures to be distinguishable under the isomorphism equivalence relation. Since, in TVLA $k \leq 2$, then under this equivalence relation, the maximal size of a set of bounded structures is $O(2^{3^p} \cdot 3^{p^2}) \subseteq O(2^{3^p})$. We call the abstract domain obtained from this technique "the relational domain".

2.2.2 The Single-Structure Method

Single structure analysis is a well known idiom in the program analysis community. It is used for a more efficient (in terms of time and space) but less precise analysis.

Conceptually, a TVLA user can assume that the number of individuals is infinite, but only a finite number of them — the *active* individuals — are visible by the analysis. The active property of individuals is maintained by a designated unary predicate ac, where $ac^S(u) = 1$ indicates that the individual u is active in the structure S. In shape analysis, $ac^S(u)$ can be one of the following,

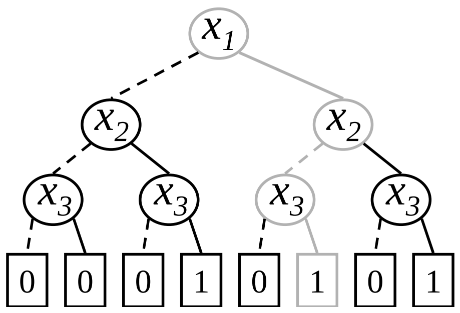
- 1 — indicating that u represents at least one concrete node in all the concrete structures represented by this structure.
- $1/2$ — indicating that u may not be present in some structures (i.e., it is possible that u does not represent a concrete node in one or more of the concrete structures represented by this structure).
- 0 — indicating that u does not represent any concrete node in any of the concrete structures represented by this structure, and can thus be discarded by the implementation.

The concept of maybe active nodes gives us a way to represent all the structures in at a CFG node in a single structure. This is done by merging them and joining the values of their predicates. When a node exists in one structure and not in the other, it becomes a maybe-active node.

Using this method can result in very imprecise analyses. This issue was addressed in [LAS00] where the method was refined by avoiding joining structures with different sets of nullary predicates. This modification limits the number of structures in a set to at most 3^k, for k nullary predicates. This method made it possible to analyze certain programs where analysis with the relational method did not terminate. However, in other cases, the analysis turned out to be still overly imprecise and more time consuming relative to the relational method. In addition, achieving precision with the single-structure method often required a modification of the operational semantics to find appropriate nullary predicates, which required human effort and made it more complex. We call the abstract domain obtained from this technique "the single-structure domain".

Arguments			Value
x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a)



(b)

Figure 2.6: (a) A truth table representation of function f. (b) A BDD representation of f. Each nonterminal tree vertex is labeled by a decision variable. A dashed tree branch denotes the case where the decision is 0 and a solid branch denotes the case where the decision is 1. The table entry $x_1 = 1, x_2 = 0, x_3 = 1, f = 1$ corresponds to the gray path in the decision tree.

2.3 OBDDs

In this section we give a very brief introduction to OBDDs. A more extensive introduction is given in [Bry86], [Bry92] and [BRB90].

Binary Decision Diagrams (BDDs) are a representation of boolean functions as a binary tree. The vertices are labeled by boolean variables, and the leaves, usually referred to as *terminals*, are labeled by boolean values. A BDD represents a boolean formula in the form of a case analysis. For example, Fig. 2.6 shows a boolean function and the corresponding BDD. For a given assignment to the variables, the value yielded by the function is determined by tracing a path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then given by the terminal vertex label.

An *Ordered* BDD (OBDD) is a normal form for BDDs where we impose a total ordering over the set of variables and require that for any vertex and either nonterminal child, their variables respect that order. In the decision tree of Fig. 2.6, the variable order is $x_1 < x_2 < x_3$.

A *Reduced* OBDD (ROBDD) is a further normal form for OBDDs obtained by repeated application of the following rules:

- **Remove Duplicate Terminals:** Eliminate all but one terminal vertex with a given label and redirect all edges into the eliminated vertices to the remaining one.
- **Remove Duplicate Nonterminals:** If nonterminal vertices u and v labeled with the same variable have $lo(u) = lo(v)$, and $hi(u) = hi(v)$ where $lo(\cdot)$ is the vertex obtained by

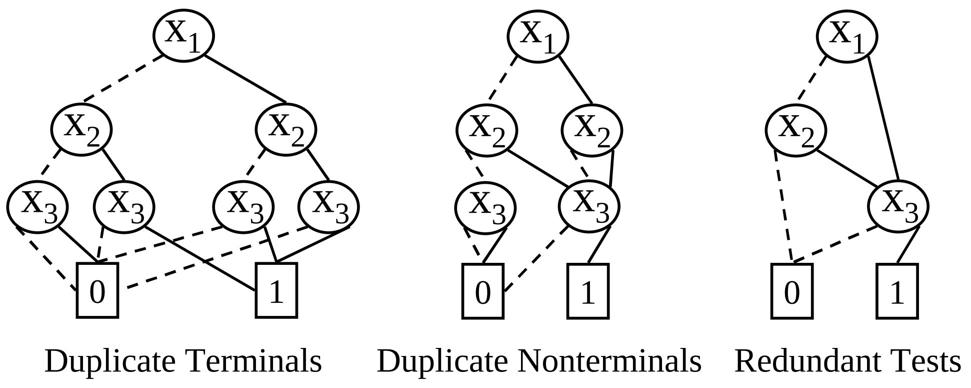


Figure 2.7: The OBDDs of f obtained by applying the reduction rules.

following the edge with a 0 decision and $hi(\cdot)$ is the vertex obtained following the edge with a 1 decision, then eliminate one of the two vertices and redirect all incoming edges into the eliminated vertex to the other vertex.

- **Remove Redundant Tests:** If a nonterminal vertex v has $lo(v) = hi(v)$, then eliminate v and redirect all incoming edges into v to the vertex $lo(v)$.

Fig. 2.7 shows an example of how the application of the rules results in an acyclic graph with fewer vertices than in the original decision tree. An extension to ROBDDs [BRB90] supplies a normal form for any number of boolean functions. The functions are represented by a multiply-rooted directed acyclic graph that allows sharing sub-graphs between all of the functions. In the remainder of this thesis we use this extension and refer to it simply as OBDD.

Chapter 3

Compactly Representing First-Order Logical Structures

This chapter addresses the problem of space consumption in first-order state representations by describing and evaluating two new structure representation techniques. The first representation uses an existing *ordered binary decision diagram* [Bry92, YBO$^+$98] (OBDD) implementation [Som98] to encode first-order structures. The second state representation combines ideas from efficient implementations of *functional maps* (where a map derived by update to another map shares substructures of the initial map) with *normalization* via *hash-consing*. While both of these core data structures are well-known, it is not obvious that they should be adaptable to, or beneficial for, representing evolving first-order structures. In addition, both of these data structures can have poor worst-case performance; thus an empirical evaluation is crucial for determining their practical benefit. Our evaluation of the new state representations indicates that they can reduce TVLA's space consumption by a factor of 4 to 10 without compromising time performance. In addition, as the number of structures manipulated by the analysis increases, the relative advantage of the new representations also increases.

3.1 Evolving First-Order Structures as an Abstract Data Type

TVLA's architecture treats first-order structures as instances of an abstract data type (ADT); this data type admits a variety of implementations. The TVLA system can then be viewed as an engine that traverses the CFG and invokes actions that update instances of the structure ADT.

We are aware of no prior work that directly addresses the issue of efficiently manipulating evolving first-order structures. However, a close examination of the operations on states performed by TVLA reveals several opportunities for significant space savings (We expect that these optimization techniques are applicable even to other state representations used for other abstract interpretations or other static analyses):

- *Sparse* data structures. For many applications, much of the state space accessible at a particular program point represents trivial values (e.g., null pointers). This fact can be exploited by avoiding explicit representation of such values.

- *Sharing* of inherited substates. The TVLA actions that model each program statement typically affect only a small portion of the program state. State representations that allow the unchanged, "inherited" portion of the state to be shared between the pre- and post-update state are therefore advantageous.

- *Normalized* (or canonical) representations for substates. Many program states contain substructures that are semantically equivalent, even though the substructures were generated by unrelated sequences of state updates. By using a canonical representation (e.g., a hash-consed tree data structure) for substates, such "serendipitous" similarity can be recognized and exploited to allow the substate to be shared. One reason why such similarity arises is that typically every program point has an associated set of invariants that hold true at all states possible at that program point. With a well-designed state abstraction, these invariants can lead to "equivalent substructures". This can lead to significant savings especially for program points inside loops, where different structures may end up with some "loop invariant substructure". Further, since nearby program points tend to have many common invariants, such "equivalent substructures" are possible even across structures associated with different program points. Another advantage of a canonic representation is that it enables efficient equality-checking of states, via a single pointer check, regardless of the size of the state representation.

- *Phase-sensitive* state representations. Structures are initially *mutable*, and may be updated as a result of applying TVLA actions. Subsequently, they become immutable, at which point they may be compared for equality with other structures, but may not be updated. The system can exploit these phase distinctions, e.g., by using a representation that admits time-efficient update for mutable structures, and a more space-efficient representation for immutable structures.

The full signature of the ADT used to represent evolving first-order structures in TVLA is depicted in Fig. B.1; of Appendix B; here we note that the ADT contains two distinct representations of structures: a mutable structure (`TVS`) and an immutable structure (`ImmutableTVS`). In addition, it defines an interface to a set of structures (`TVS_SET`), which contains only immutable structures. The `add` operation for `TVS_SET` returns true if the structure being added is not an element of the set, and false otherwise; this entails checking whether the added structure is *isomorphic* to any structures already in the set. (The isomorphism check can be done in polynomial time for *blurred* structures since a blurred structure has at most one node with any given canonic name.)

The distinction between mutable and immutable structures is noteworthy. Mutable structures require a representation that allows operations such as predicate updates, and addition and removal of nodes to be done efficiently. On the other hand, the only operation performed on an immutable structure is the isomorphism test implicitly required by the `add` operation of `TVS_SET`. This distinction allows us to use a normalized, space-efficient representation for immutable structures. This property is especially important because immutable structures last for the duration of the entire analysis, whereas the mutable structures can be discarded after they are popped from the stack and processed (See Fig. 2.4).

3.2 TVS Representations

In this section, we describe three different implementations of the TVS ADT.

3.2.1 Base Representation

We first describe the representation used in the public release of TVLA (TVLA version 0.91). We will refer to this as the Base representation, and use it as a baseline against which we compare the other, newer, representations.

The value of an unary (or binary) predicate p in a structure is represented by a HashMap (a hashtable based implementation of a map available in the Java Collections Framework) that contains an entry for every node i (or node pair (i, j) in the case of binary predicates) for which the value of $p(i)$ (or $p(i, j)$ respectively) is nonzero. The predicate p itself is said to have a *nonzero* value if the value of $p(i)$ (or $p(i, j)$ in the case of binary predicates) is nonzero for at least one node i (or node pair (i, j)). The value of a nullary predicate is represented by a single Kleene value.

The value of a structure itself is represented by a HashMap (which we will refer to as the first-level map) that contains an entry for every predicate p that has a nonzero value. The entry corresponding to a predicate p contains the value of p (a HashMap for non-nullary predicates and a Kleene value for nullary predicates), as well as a boolean flag, which is used to achieve some amount of sharing between the representations of different structures as explained below.

When a structure is copied, its first-level map is duplicated, but the HashMaps representing the values of (non-nullary) predicates are shared by the original and new structure. The boolean flag associated with these predicate values is set to indicate this sharing. When the value of a predicate with a shared representation needs to be updated, the underlying shared HashMap is duplicated before the update is performed. The associated boolean flag is also reset at this point to indicate the absence of sharing.

The universe of a structure is implemented using a HashSet (a hashtable based implementation of a set) plus a boolean flag to implement a similar "copy-on-write" scheme for the universe also.

3.2.2 OBDD Representation

We now describe a new implementation of the TVS ADT, which we will refer to as the OBDD representation. This is a phase-sensitive representation, where mutable structures are represented using the Base representation, but immutable structures are represented using OBDDs as explained below.

Our representation uniformly models all predicates as if they were binary predicates. A unary predicate p is represented as if it were a binary predicate by translating references to $p(u)$ into a reference to $p(0, u)$. Nullary predicates are modelled using binary predicates by translating references to p into references to $p(0, 0)$. The representation uses a set P of boolean variables to identify predicates, a set N_1 of boolean variables to identify the first argument (a node) of the predicate, a set N_2 of boolean variables to identify the second argument (a node) of the predicate, and a special boolean variable $v_{1/2}$, used to extend the representation to 3-valued logic, as explained below. A 3-valued structure may then be thought of as a boolean function over these variables, one that returns the value of the predicate identified by P for the tuple of nodes identified by N_1 and N_2. Since the value of the predicate is a Kleene ({0, 1, 1/2}) rather than a boolean, the variable $v_{1/2}$ is used to encode the third value. We represent the 3-valued structure using a corresponding OBDD with the variable ordering $N < P < \{v_{1/2}\}$, where N denotes the sets of variables N_1 and N_2 interleaved.

One of the goals of the representation is to ensure that (*blurred*) *isomorphic* structures end up with the *same* OBDD representation. We ensure this as follows. Recall that a node's canonic name is defined to be the sequence of values of the unary predicates for that node. When the OBDD representation of a structure needs to be created, the nodes of the structure are first sorted by canonic name and then contiguously numbered from 0 on. This guarantees that isomorphic structures will be represented by the same OBDD. Since only immutable structures are represented with OBDDs, the sets of variables N_1 and N_2 actually needed to identify nodes is determined when a structure is converted to an OBDD representation by choosing $|N_1| = |N_2| = \ulcorner \log |U^S| \urcorner$ OBDD variables.

For expository purposes, we use a simpler analysis[1] than that of the running example. Consider the program depicted in Fig. 3.1, which prints all the elements of a singly-linked list. Let us say that we wish to confirm the (here obvious) assertion that the `printf` statement is never executed when the variable `x` has the value `NULL`. A very simple heap *shape analysis* [SRW02] can be used for this purpose, which uses just the predicates sm, x, y and n with the same meaning as described in Table 2.2.

[1]This analysis generates a very small number of simple structures, making it possible to demonstrate our ideas more concisely.

```
/* list.h */            /* print.c */
typedef struct node     #include "list.h"
{                       void print_all(L y) {
  struct node *n;         L x;
  int data;               x = y ;
} *L;                     while (x != NULL) {
                              /* assert x != NULL */
                              printf("elem=%d ", x->data);
                              x = x->n;
                          }
                        }
```
(a) (b)

Figure 3.1: (a) Declaration of a linked-list data type in C. (b) A C function that prints all the elements of the list pointed to by parameter `y`.

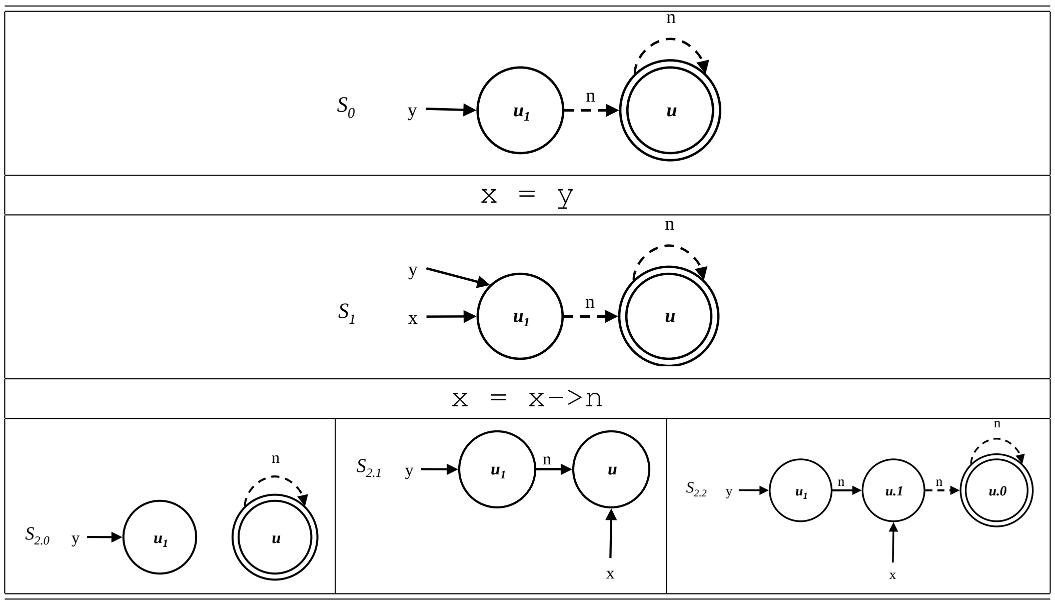


Figure 3.2: Structures generated by abstract interpretation of the first iteration of the loop in the `print_all` function.

The structures generated by abstract interpretation of the first iteration of the loop body of the `print_all` function are depicted in Fig. 3.2. After two iterations, the abstract interpretation of the `print_all` function reaches a fixpoint, with an output state containing the same set of structures ($\{S_{2.0}, S_{2.1}, S_{2.2}\}$) generated after the first iteration.

Fig. 3.3 shows the OBDD representation of the structures S_0, S_1 and $S_{2.2}$ from Fig. 3.2. The OBDD nodes labelled $u1$ or $\langle u, u.1\rangle$, for example, represent corresponding tuples of nodes. Paths from the root to these OBDD nodes correspond to the node tuple boolean variables. Paths from these OBDD nodes to the terminals 0 and 1 correspond to the predicate variables. For example, the path 010, starting from S_0, conveys the fact that in S_0 the predicate x evaluates to 0 on for the node u_1. The path 0011 starting from S_1 and stopping at the $1/2$ node conveys the fact that the sm predicate evaluates to $1/2$ on the node pair (u, u) in the structure S_1.

This figure illustrates the two kinds of sharing described earlier. The difference between the OBDD representing S_0 and the OBDD representing S_1 are the two grayed nodes, which follow the path 010 starting from S_0. This reflects that the only difference between the structures is in the interpretation of the predicate x for the node u_1, leading to *inherited* sharing between the two OBDDs. Non-inherited sharing can be seen by observing that the node annotated by u, $\langle u, u\rangle$ is shared by both the OBDD of S_1 and $S_{2.2}$. This reflects the fact that the node u that represents the tail of the linked list in S_1 and the node $u.0$ that represents the tail of the list in $S_{2.2}$ have the same canonic name. This allows the two structures to share the values of sm and n for this node. This sharing could easily be missed by implementations relying solely on inheritance-based sharing. The more a structure S is mutated to produce a structure S' less the inherited sharing between them, even if they contain many similar sub-structures. The OBDD representation is insensitive to the scenario by which S' evolved from S and therefore manages to exploit these similarities. Also notice that this program has a simple loop invariant that leads to sharing in the OBDD representation of the structures arising after the statement `x = x->n` ($S_{2.0}$, $S_{2.1}$ and $S_{2.2}$.) In every iteration of the loop, the node u_1, which represents the head of the list has the predicate values $x(u_1) = 0$, $y(u_1) = 1$, $sm(u_1) = 0$ and $n(u_1, u_1) = 0$, which enables all of the structures to share these values.

3.2.3 A Functional Representation

We now describe a representation of 3-valued first order structures we refer to as a functional representation. This implementation utilizes techniques similar to those used in OBDDs, but in the context of a different data structure, namely maps. This makes it more convenient, for instance, to implement the higher level TVLA operations, without having to encode them in terms of OBDD operations.

We assume that the nullary predicates are numbered from 0 to n_0, that unary predicates are numbered from 0 to n_1, and that all binary predicates are numbered from 0 to n_2. We assume that every node in a structure is assigned a unique integer value. However, unlike in the case

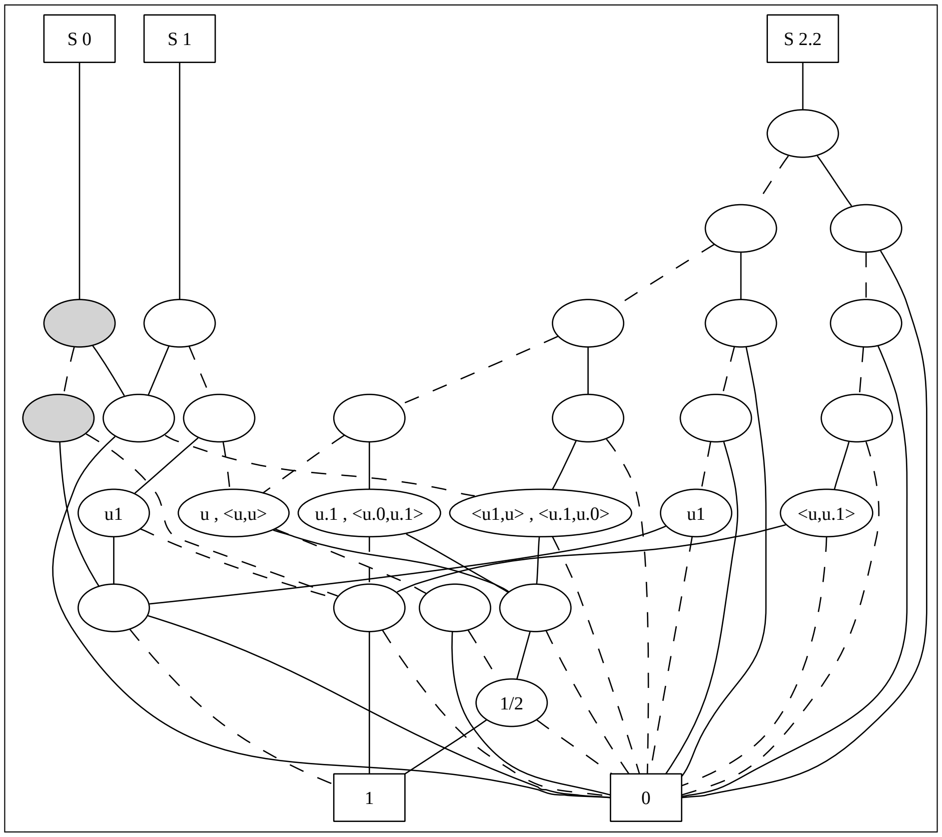


Figure 3.3: The OBDDs representing the 3-valued structures in Fig. 3.2. Dashed edges denote 0, and solid edges denote 1. The node ordering imposed by the canonic names is $u < u_1$ for S_0 and S_1, and $u.0 < u.1 < u_1$ for $S_{2.2}$. The predicates are numbered as follows: $sm : 0$, $x : 1$, $y : 2$, $n : 3$.

of predicates, the representation places no limit on the number of nodes in a structure. Further, since the set of nodes in a structure can change dynamically, the nodes in a given structure are not required to be assigned contiguous numbers. The set of nodes in a structure is implemented as a linked list, which is manipulated in a functional style to allow sharing between the representations of different structures.

The values of nullary predicates in a structure is represented by a map from integers in the range $[0 : n_0]$ to Kleene. We use a tree-based functional data structure[2], which we will refer to as a *flik*, to implement such a map. A flik is either a *leaf*, capable of storing upto some fixed number l of Kleene values, or a *branch*, consisting of a fixed number a of children (each of which is a flik), as well as an integer *size* field.

A map from $[0 : i]$ to Kleene can be implemented using a single leaf for any $i < l - 1$. For $i \geq l$, we require a branch flik. A branch of size s implements a map from $[0 : s - 1]$ to Kleene by splitting the interval $[0 : s - 1]$ into a equal subintervals, each managed by a corresponding child. A subinterval where all the Kleene values are 0 need not be represented by a corresponding child (i.e., the child's value will be null). Further, a branch may be replaced by its first child if all of its other children are null.

A lookup or an update operation can be done with this representation in $O(\log N)$ time, where N is the size of the domain of the map. Note that the implementation is functional: an update returns a new tree and does not modify the original tree. As with all functional data structures, the new tree will share the unmodified parts of the old tree with the old tree.

We adapt the above data structure to represent maps from integers to an arbitrary set, by changing the representation of a leaf to store upto some fixed number l' of object references. We refer to this modified data structure as an *intmap*. An intmap can also be adapted to represent a map from *pairs of integers* to some arbitrary set, by first utilizing any suitable encoding function that maps every pair of integers to a unique integer. We refer to such an adaptation as an *intpairmap*. (Our implementation uses an encoding function that maps a pair (i, j) to the integer $(i + j) * (i + j + 1)/2 + i$, but other encoding schemes are possible.)

The values of unary predicates in a structure is represented using a *two-level map*: this consists of an intmap (the first-level map) that maps every node i in the structure's universe to a flik (the second-level map) that maps every unary predicate p's number to the value of $p(i)$ in the universe. The values of binary predicates is also represented by a two-level map: the first-level map (an intpairmap) maps a pair of individuals (m, n) to a second-level map (a flik) that maps a binary predicate p's number to the value of $p(m, n)$ in the structure.

An alternative implementation would be to swap the order in which nodes and predicates are used as arguments of two-level maps. We chose the above ordering as it makes operations such as *blur* and isomorphism testing more efficient. In particular, these operations often check for

[2]The data structure may be logically viewed as a tree, though sharing of subtrees can lead the actual representation to be a dag, just as in an OBDD.

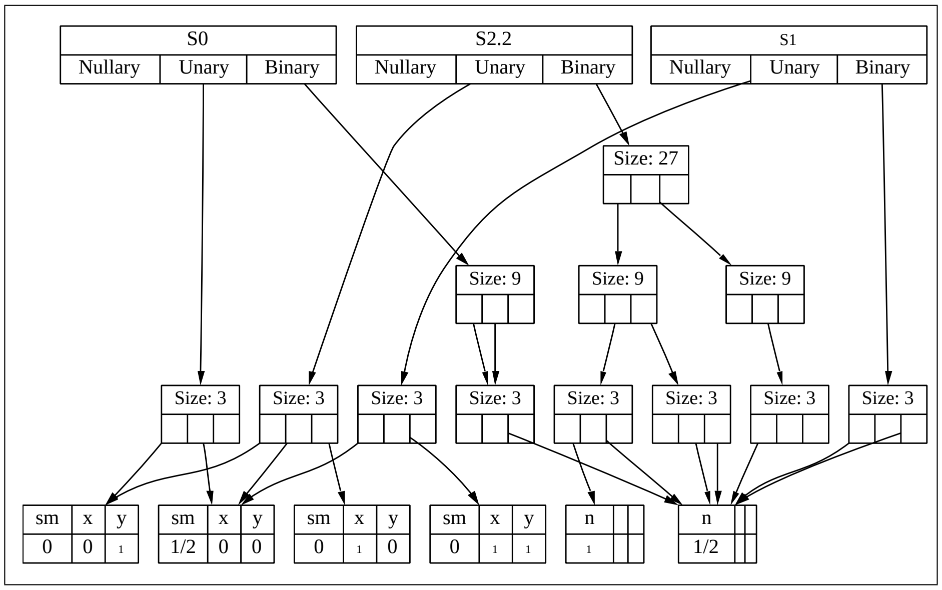


Figure 3.4: The functional representation of the 3-valued structures in Fig. 3.2. Lists representing the universes are not shown in the figure. Nodes u_1 and u of structure S_0 are assigned numbers 0 and 1, respectively. Nodes u_1 and u of structure S_1 are, however, assigned numbers 1 and 0 respectively. Normalization causes this node number reassignment because the canonic name of node u_1 has changed. Nodes u_1, $u.0$, and $u.1$ of structure $S_{2.2}$ are assigned numbers 0, 1, and 2 respectively. The lowermost layer represents the second-level maps, while the layers above represent the first-level maps.

a pair of nodes m and n if the set of all unary predicates' values for m equal the set of all unary predicates' values for n. Our ordering enables us to perform such a step by looking up the flik f1 that represents all unary predicates' values for m, and looking up the flik f2 that represents all unary predicates' values for n, and then by comparing fliks f1 and f2 for equality.

Fig. 3.4 shows a representation of the structures S_0, S_1 and $S_{2.2}$ from the example of Fig. 3.2. This representation uses a value of 3 for all the parameters a, l, and l'. Notice that even though there are a total of 7 nodes in the 3 structures, the representation requires only 4 fliks to represent their canonic names (i.e., the value of all unary predicates for these nodes).

Since the underlying implementation is functional, a 3-valued structure can be copied using a *shallow* copy of the pointers to the data structures described above. The resulting copy completely shares the underlying representation with the original. If this copy is subsequently modified via some sequence of operations, it will continue to share unchanged "inherited" portion of the representation.

Representation of Immutable TVS

Our implementation also utilizes *normalization*, which replaces distinct occurrences of equivalent data (which may be a fragment of the representation) by a unique or *canonical* representative. Normalization has two benefits: it increases the sharing between the representations of different structures and reduces the space requirements; it also makes it possible to check for equality (isomorphism) between structures more efficiently, e.g. through a pointer equality comparison. We normalize structures when they become immutable.

Normalization is done by first sorting nodes, based on their canonic names, and renumbering them from 0 on, and by then applying *hash-consing*: a hash table is used to store canonical representatives; a structured object obj is normalized by first recursively normalizing its substructures and by then checking for the occurrence in the hash table of any object equivalent to obj; if such an object exists, obj is replaced by that object; otherwise, obj is added to the hash table.

Using Delayed Normalization

Our implementation does not attempt to keep all structure representations normalized all the time. Recall our earlier discussion about the "lifetime" of a structure. Usually, a structure is born as a copy a normalized structure. During its active life (growth phase), no attempt is made to keep the structure's representation normalized. The structure is normalized only at the end of its growth/active phase.

3.3 Empirical Evaluation

In this section we present an empirical evaluation of the representations described in Section 3.2. Though we present timing results as well, our focus is primarily on evaluating the space requirements of the different representations. The experiments were conducted using TVLA version 0.92, running with SUN JDK 1.3, on a 1 GHZ Intel Pentium Processor machine with 1 GB RAM. The OBDD representation was implemented with the CUDD [Som98] package.

Benchmarks

The benchmarks used in the experiments are explained below. The `CA` benchmark performs "cleanness analysis" [DRS00] in a C program implementing the instruction selection phase of the Tiger educational compiler [AG98]. The `GC` benchmark [RSW01] involves a partial verification of the mark phase of a mark-and-sweep garbage collector. The `JFE` and the `Kernel` benchmarks are both instances of the Concurrent Modification Problem (CMP) described in [RWFS01]. CMP requires identifying a specific type of misuse of Java Collection Classes. Finally, the `MA` benchmark [NNS00] verifies certain safety properties of a packet router in the mobile ambient calculus [CG98]. This is a particularly challenging verification problem since it describes a dynamically evolving non-deterministic program and yet TVLA is able to show that the packet resides in one router.

Results

Table 3.1 presents actual time and space statistics for running TVLA on the benchmark programs using each of our three implementations. These results indicate that both the OBDD implementation and the Functional implementations consume significantly less memory than the Base implementation. In the case of the OBDD implementation the table also shows how much memory was used by the CUDD package (which measures the memory used by the immutable structures that are represented as OBDDs) which is significantly less than the total memory used by the system, which additionally includes the memory used by the mutable structures in the Base representation. This shows the potential space reduction possible using a pure OBDD representation. The timing results show that the three implementations are comparable in performance.

Measuring Space Usage Via Instrumentation

Several factors confound a comparison of the different representations by measuring actual memory usage. In particular, actual memory usage is affected by the factors such as the programming language used, the runtime system used (e.g., a 16 byte object overhead in some Java implementations), libraries used, and the extent to which the implementation was carefully

Table 3.1: Time and space consumption of the representations. Time is measured in seconds and space in megabytes. The **Struct.** column denotes the number of structures. Recall that the OBDD representation is a phase-sensitive representation that uses the Base representation for mutable structures and OBDDs for immutable structures. In this case, we show both the memory used for representing all structures (the column labelled "Total Space") and the memory used for representing immutable structures (the column labelled "Immutable Space").

Benchmark	**Struct.**	**Base**		**OBDD**			**Func.**	
		Time	**Space**	**Time**	**Total Space**	**Immutable Space**	**Time**	**Space**
`CA`	40,000	1,861	168.2	1,874	32.6	12.6	4,567	12.9
`GC`	189,772	3,822	402.8	2,686	192.1	16.5	2,446	51.6
`JFE`	10,424	27	12.8	28	5.8	1.1	13	5.5
`Kernel`	6,079	29	22.7	40.2	6.8	3.4	22.4	16.7
`MA`	20,000	3,724	187.7	3,758	109	8.6	4,489	9.6

engineered. Hence, the actual memory usage is not a very accurate indicator of the memory that a representation actually requires.

Therefore, we also instrumented our implementations to compute the actual number of objects of different type used by the different representations. From these counts we estimated the memory that would be required by a reasonable implementation, not counting overheads imposed by different runtime systems. Table 3.2 presents the statistics produced by our instrumentation. The table also reports two "metrics" (dense and sparse), which are different measures of the amount of information contained in the actual set of structures produced by the analysis. The *dense* metric is the space required to represent the structures, storing every predicate value explicitly in a bit-packed fashion, using two bits per predicate value. The *sparse* metric is the space required to store just the non-zero predicate values, using 4 bytes to identify each non-zero predicate value. The OBDD metric is obtained by multiplying the maximum number of live OBDD nodes that existed during the analysis (which is provided by the CUDD package) by 20, assuming that an OBDD node can be implemented using 20 bytes. The Functional metric is obtained by multiplying the number of objects used by the representation by 24 (as all objects in this implementation require 24 bytes or less). The Base representation metric is similarly computed, using appropriate sizes for the different kinds of objects used by this representation.

These results too indicate that our OBDD and Functional representations do very well and are better than the Base representation by an order of magnitude.

Asymptotic Trends

Since we are interested in a scalable TVS representation, we also measured how space consumption varies over the duration of the analysis. Specifically, we computed the instrumentation-based

Table 3.2: Space counters for different representations. These counters indicate the number of bytes required to represent the structures and are computed as explained in Appendix C. The **Struct.** column denotes the total number of structures produced by the analysis.

Benchmark	**Struct.**	**Dense**	**Sparse**	**Base**	**OBDD**	**Func.**
`CA`	40,000	7,473,737	11,053,352	22,516,937	2,302,140	1,749,384
`GC`	189,772	9,769,618	32,722,016	41,835,001	4,268,780	7,288,032
`JFE`	10,424	49,172,345	382,156	1,201,570	181,940	300,336
`Kernel`	6,079	64,768,292	799,604	2,292,436	420,520	315,168
`MA`	20,000	18,866,654	8,170,412	17,077,413	496,960	724,152

Table 3.3: Abstract counters used to represent TVS at different execution points of the iterative procedure for the mobile ambients benchmark. We sample every 2000 structures.

Sample	**Dense**	**Sparse**	**Base**	**OBDD**	**Func.**
1	932	395	978	89	70
2	938	399	898	60	52
3	938	400	899	49	48
4	937	403	917	51	47
5	939	405	955	50	45
6	941	406	963	48	44
7	943	407	966	46	41
8	943	408	921	40	36
9	943	408	884	36	32
10	943	409	854	32	29

space usage estimate periodically. Table 3.3, Table 3.4 and Table 3.5 show that as the analysis proceeds, the average size of the structure increases, as measured by both the dense metric and the sparse metric. This is consistent with our expectations, since the state space exploration typically starts examining more complex structures, with more individuals, as time proceeds. However, it may be seen that the space required to represent an average structure, decreases for the OBDD and Functional representation, indicating that the benefits of sharing increase as more structures are produced.

Table 3.4: Abstract counters used to represent TVS at different execution points of the iterative procedure for the cleanness analysis benchmark. We sample every 5000 structures.

Sample	**Dense**	**Sparse**	**Base**	**OBDD**	**Func.**
1	155	228	505	74	50
2	154	252	531	74	53
3	165	261	539	68	48
4	168	266	547	68	47
5	173	267	554	65	46
6	186	271	561	61	44
7	188	273	565	60	44
8	187	276	563	58	44
9	190	278	561	56	43
10	192	281	564	54	42

Table 3.5: Abstract counters used to represent TVS at different execution points of the iterative procedure for the garbage-collection benchmark. We sample every $10,000$ structures.

Sample	**Dense**	**Sparse**	**Base**	**OBDD**	**Func.**
1	46	148	162	24	37
2	49	157	209	32	44
3	49	155	204	30	41
4	48	153	190	25	38
5	49	160	215	28	40
6	50	165	203	24	38
7	51	171	193	21	36
8	51	171	193	23	36
9	51	173	211	25	38
10	52	174	225	26	40

Chapter 4

Tuning Abstraction to Improve Performance

In this chapter we are interested in finding abstractions that require lower analysis costs than the costs required by the relational method. The main idea is to find opportunities for compacting sets of structures, without compromising the goals of the analysis. The new (compacted) sets yield a conservative solution to the analysis, but can potentially reduce the quality of the approximation. We empirically evaluate the abstraction methods over a set of benchmarks and show that, in practice, they give the same precision as the relational method but require significantly less time and space resources.

To simplify the discussion, we will ignore nullary predicates in the rest of this chapter (their treatment is straightforward).

Benchmarks

The benchmarks used throughout this chapter are explained below. The `GC` and the `MA` benchmarks were introduced in Chapter 3. The `reverse` benchmark is our running example. The `merge-sorted` benchmark performs partial verification of a procedure that merges two sorted linked-lists in order to show that it returns a sorted linked-list. The `insert-sort` and `bubble-sort` benchmarks perform partial verification of sorting algorithms on singly-linked lists.

4.1 The Mathematical Framework

The embedding relation between a pair of three-valued structures defines a partial order that can naturally be extended to sets of structures. Given two sets of three values structures XS_1 and XS_2, the partial order is defined as $XS_1 \sqsubseteq XS_2$ if for every structure S in XS_1 there exists a structures S' in XS_2 such that $S_1 \sqsubseteq S_2$.

We are interested in transformations (denoted by T) that operate over sets of 3-valued logical structures (denoted by x) and obey the following rules:

1. **Compacting** : $|T(x)| \leq |x|$
2. **Conservative** : $T(x) \sqsupseteq x$

Transformations that obey these rules are a spacial case of the widening operator [CC79] that ignores previous iterates. All of the set compaction techniques that are described in this chapter obey the rules mentioned above.

4.2 Exploiting Embedding to Optimize the Relational Method

As mentioned in Section 2.2, the single-structure method can lead to imprecise analyses. We therefore choose to start with the relational method and seek techniques and heuristics to find more efficient methods.

Observation 4.2.1 *A set of 3-valued structures, obtained from existing TVLA abstraction methods, can contain two structures S and S' such that $S \sqsubseteq S'$, even though it is sufficient to keep only S'. S is* superfluous *and removing it will not lead to a loss of soundness, since S' already represents a superset of the 2-valued structures that S represents.*

Observation 4.2.1 gives us a way to reduce the size of structure sets. By comparing every pair of structures in a set, we can decide which structures should be kept and which structures can be safely discarded.

Example 4.2.1 Fig. 4.1 shows an example of two structures that arise during the analysis of the running example with the relational method. Structure S_1 represents a store that arises when `reverse` is invoked with a list containing exactly three elements. Structure S_2 represents the case where `reverse` is invoked with a list contains any number of cells greater than two. Structure S_1 is a special case of S_2 where node u_1 represents exactly one list cell. In this case, the optimization we seek would remove S_1 and keep S_2.

The following lemma shows that structure embedding is generally a very tough problem for which there is no known polynomial time solution.

Lemma 4.2.2 *GRAPH ISOMORPHISM $\leq_\rho$ STRUCTURE EMBEDDING*

Proof: Appears in Appendix D.

Although, in general, the embedding problem is hard, there are some restricted cases where an efficient solution exists.

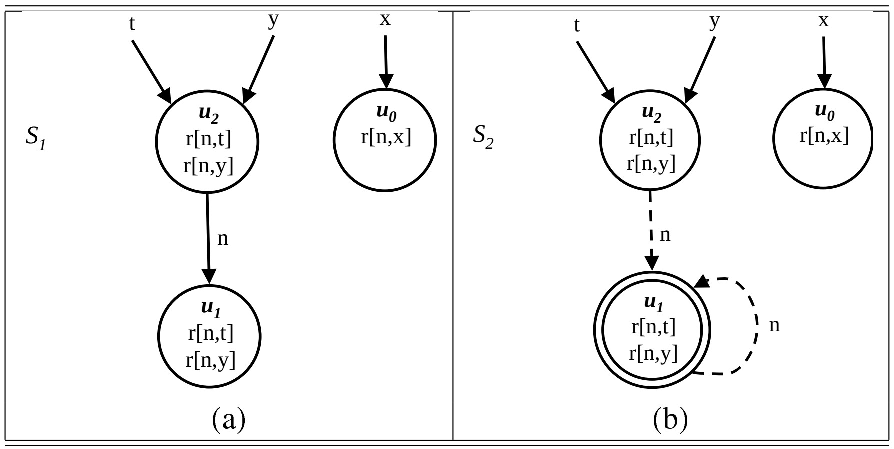


Figure 4.1: (a) A structure S_1 that represents the case of reversing a singly-linked list with three cells. (b) A structure S_2 that represents the case of reversing a singly-linked lists with **at least** three cells. Structure S_2 embeds structure S_1 with the function mapping node u_i in S_1 to node u_i in S_2 for $i \in \{0, 1, 2\}$.

Definition 4.2.3 *We say that a structure S is **definite** if all abstraction predicates evaluate to definite values for all nodes.*

Note that the *blur* operation is closed under this definition, since it never alters that values of abstraction predicates.

Lemma 4.2.4 *Suppose that both S_1 and S_2 are bounded, definite and $S_1 \sqsubseteq^f S_2$ then S_1 and S_2 contain the same set of canonical names. That is, f is one-to-one and for every individual $u_1 \in U^{S_1}$ there is exactly one individual $u_2 = f(u_1) \in U^{S_2}$ such that for every unary abstraction predicate p we have $p^{S_1}(u_1) = p^{S_2}(u_2)$.*

Proof: Suppose, by contradiction, that f is not one-to-one, so $|U^{S_1}| > |U^{S_2}|$. Since f is surjective there are two individuals $u, v \in U^{S_1}$, such that $f(u) = f(v) = w$. From the assumption that $S_1 \sqsubseteq^f S_2$ and the definition of embedding we know that, for every unary abstraction predicate p, the following relations hold : $p^{S_1}(u) \sqsubseteq p^{S_2}(w)$ and $p^{S_1}(v) \sqsubseteq p^{S_2}(w)$. From the assumption that both structures are definite, we get that $p^{S_1}(u) = p^{S_2}(w) = p^{S_1}(v)$. Therefore, u and v have the same canonical name. This contradicts the assumption that S_1 is bounded (canonical names are unique in bounded structures). The conclusion is that f is one-to-one and every pair of corresponding individuals $u_1 \in U^{S_1}$ and $f(u) \in U^{S_2}$ share the same canonical name.

Using Lemma 4.2.4 we optimistically attempt to reduce the size of a set of structures with the algorithm shown in Fig. 4.2. The algorithm directly applies the conditions of the lemma to every pair of structures and merges pairs of structures if it can resolve that one is embedded in the other. In cases where the conditions of the lemma do not hold, no compaction is done.

```
compact(in : set of structures) {
       change = true
       while (change) {
             change = false
             if exist $S_1, S_2 \in$in s.t. $S_1 \neq S_2 \wedge$ compatible($S_1, S_2$) {
                in' = (in - $\{S_1, S_2\}$)$\cup$\{merge($S_1, S_2$)\}
                change = true
                in = in'
             }
       }
}

// check the conditions of Lemma 4.2.4
compatible($S_1$ : structure, $S_2$ : structure) : boolean {
      if $canonicNames(S_1) = canonicNames(S_2)$ {
         let $f : U^{S_1} \to U^{S_2}$ be the one-to-one map between
         individuals with the same canonical name.

         // check whether $S_1 \sqsubseteq S_2$
         if for every predicate $p$ of arity $k$ and $u_1, \ldots, u_k$
         $p^{S_1}(u_1, \ldots, u_k) \sqsubseteq p^{S_2}(f(u_1), \ldots, f(u_k))$
            return true

         // check whether $S_2 \sqsubseteq S_1$
         if for every predicate $p$ of arity $k$ and $u_1, \ldots, u_k$
         $p^{S_2}(u_1, \ldots, u_k) \sqsubseteq p^{S_1}(f(u_1), \ldots, f(u_k))$
            return true
      }
      return false
}
```

Figure 4.2: Reducing the size of a set of structures by merging pairs of "compatible" structures.

The algorithm requires polynomial costs, and for the actual implementation we employed several optimizations that enabled a virtually linear time algorithm (by using hashing). The complexity analysis and actual implementation details are not particularly interesting and are therefore omitted.

Fortunately, for the benchmarks we experimented with, virtually all of the structures produced by the analysis were definite (in fact, the only case where the structures were not definite was when the input structures were not definite.) Thus, our optimization payed off and the algorithm was able to remove all embedded structures. Table 4.1 shows the improvement gained by using this method. This experience indicates that shape-analysis typically produces a considerable amount of superfluous structures. For example, in the `bubble-sort` benchmark about half of the explored structures were found to be superfluous and removing then reduced time and space by approximately the same factor. In the `GC` benchmark the improvement is even more dramatic — the number of explored structures was reduced by more than an order of magnitude with similar effect on the time and space requirements. This technique also allows the `MA` benchmark to terminate, in contrast to the analysis that used the relational method (that was aborted after running for more than 4 hours).

Table 4.1: Comparison of the results obtained with the relational method and the relational method with the embedding optimization. #strucs indicates the number of structures explored by the analysis, the time is measured in seconds and space is measured in Mb. The analysis of the `MA`benchmark did not terminate after 14,400 seconds.

Benchmark	**Relational Method**			**Optimized Relational Method**		
	#strucs	**time**	**space**	**#strucs**	**time**	**space**
`MA`	>38,784	>14,400	>12.8	746	577	2.5
`reverse`	70	1	0	70	1	0
`merge-sorted`	1240	25	4.1	575	13	1.5
`insert-sort`	2,828	57	9.9	933	19	2.1
`bubble-sort`	3,012	58	9.9	1430	32	3
`GC`	189,772	3,231	413.7	11,363	489	19.5

4.3 A More Compact Domain Based on Canonical Names

We now describe a method for obtaining a new abstract domain that is more compact than the one described in the previous section, and then proceed to show that in practice it is as precise as the domain obtained with the relational method.

The new method attempts to adopt the principles that are used to define the abstraction function over single structures to sets of structures. The abstraction of a single structure is obtained by the *blur* operation. As noted before, the *blur* operation transforms a definite structure to another definite structure but can modify the values of non-abstraction predicates, thus keeping track of one group of predicates (abstraction predicates) more precisely than the other (non-abstraction predicates). We apply this principle to sets of structures by merging structures that contain the same set of canonical names. Thus, the method discussed in this section uses a condition for merging structures that is more liberal than the condition used in the previous section. When two structures are found to be compatible for merging, we use their canonical names to obtain a one-to-one mapping between the individual of both structures and join the values of the non-abstraction predicates. The resulting structure preserves the values of the abstraction predicates, so the result of merging two definite structures is also a definite structure.

Example 4.3.1 Consider the structures produced during the analysis of the `GC` benchmark that are shown in Fig. 4.3. Notice that the merged structure, shown in Fig. 4.3(c) represents the same set of concrete structures as the structures in Fig. 4.3(a) and Fig. 4.3(b). This is because the only difference between the two original structures is in exactly one predicate value (that of $left$). In such a case, either of the original structures is a special case of the merged structure where the indefinite predicate value is refined to a definite value. In general, a set that contains the pair of original structures is more precise than a set that contains just the merged structure, but an analysis that uses the *best (induced) abstract transformer* [CC79] does not distinguish between

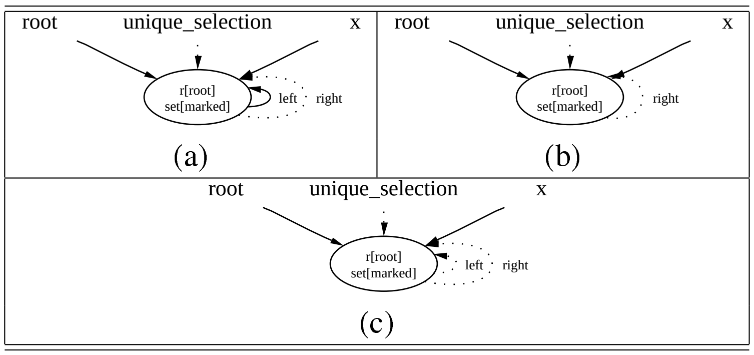


Figure 4.3: (a) A structure S_1. (b) A structure S_2 where the only difference is in the interpretation of the predicate $left$. (c) The result merging S_1 and S_2 where the interpretation of $left$ is a non-definite value.

Table 4.2: Comparison of the results obtained with the relational method with the embedding optimization and the domain obtained from the canonical names method. #strucs indicates the number of structures explored by the analysis. Time is measured in seconds and space is measured in Mb.

Benchmark	**Optimized Relational Method**			**Canonical Names Method**		
	#strucs	**time**	**space**	**#strucs**	**time**	**space**
`MA`	746	577	2.5	327	178	1.4
`reverse`	70	1	0	70	1	0
`merge-sorted`	575	13	1.5	319	6	1
`insert-sort`	933	19	2.1	932	18	2.1
`bubble-sort`	1430	32	3	792	15	1.9
`GC`	11,363	489	19.5	1,128	9	1.9

the two cases. Although the transformer used by current TVLA analyses is not the best one, it seems to be very close in practice and therefore merging structures as shown in this example does not lead to loss of precision. This example is important, since many of the structures mergers that occurred in our benchmarks were conducted for a pair of structures separated by exactly one predicate value, which helps explain why the precision of the analyses did not deteriorate.

We conducted experiments to evaluate the performance of the new method with respect to the relational method with the embedding optimization. Table 4.2 contains the results, which show a dramatic improvement over the method that exploits embedding. The most noticeable improvement occurred for the `GC` benchmark where almost every resource metric improved by two orders of magnitude. It seems that the enormous amount of structures produced by the relational analysis of the `GC` benchmark is due to an exponential blow-up caused by binary predicates.

We may be tempted to think that maintaining precision of non-abstraction predicates is completely wasteful and that the same results could be achieved by simply setting their value to $1/2$ at the end of every action. This would cause structures that were distinguishable by non-abstraction predicates to become isomorphic and merged by the relational domain. This is not the case. Doing so usually results in a deterioration of the overall precision. For example, in the `GC` benchmark, the analysis terminates after exploring 17806 structures (as opposed to 1129) in 220 seconds (as opposed to 8.5 seconds) but results with 22 false-alarms. This also demonstrates how loss of precision can lead to a less efficient analysis.

4.4 Using a Staged Analysis to Localize the Abstraction

The abstraction methods presented so far, use the same abstraction for every input program. In this section, we look for a method that adapts itself to every given input program. Our strategy is based on selecting an appropriate subset of the set of abstraction predicates for every CFG node. The modified abstraction is more coarse than the original one, but hopefully it is customized to the input program in such a way that the overall level of precision is the same.

The technique we use for selecting the subsets of abstraction predicates is inspired by compiler optimizations that employ a live-variables analysis. It is true that changing the value of a dead variable does not affect the semantics of a program. This allows, for example, compilers to allocate the same register to different variables that are never live simultaneously. We can think of the input to TVLA as a program written in the language of first-order logic with transitive closure. Recall that the goal of a live-variables analysis is to (conservatively) find, for each CFG node, which variables may be later used before they are modified when execution leaves it along some path. We adopt this goal by thinking of predicates as variables and of TVLA actions statements in the corresponding programming language. By performing a variant of live-variables analysis on the predicates of the program, we can assess whether or not the values of a predicate are to be kept precisely at any CFG node. We use the results of the analysis by marking predicates as abstraction in the CFG nodes where they are live and as non-abstraction for the other nodes.

The revised analysis is staged. The first phase selects the abstraction predicates for each CFG node and the second phase works as before, except that when abstraction is performed it is done relative to the set of abstraction predicates that correspond to the CFG node where the processed structures are being stored.

A backward liveness analysis requires an association of `USE` and `DEF` sets for every CFG node. The `USE` set describes the set of facts that are referenced by a statement and the `DEF` set describes the set of facts that are modified by a statement. In order to define the liveness analysis for TVP specifications, we proceed by describing how to extract the `USE` and `DEF` sets from

TVLA actions[1].

A TVLA action is made of the following components:

- **Precondition formulae.** The predicates that appear in the formulae are added to the action's `USE` set.
- **Focus formulae.**[2] The predicates that appear in the formulae are added to the action's `USE` set.
- **Update formulae.** The predicates that appear on the right hand side of update formulae are added to the action's `USE` set and the predicates that appear on left hand side of the update formulae are added to the action's `DEF` set.
- **Message formulae.**[3] The predicates that appear in the formulae are added to the action's `USE` set.

Another implicit component of an action is the set of constraints formulae that are used by the *Coerce* algorithm. The predicates that appear in these formulae are not considered, since they are not part of the concrete semantics. Focus formulae, on the other hand, are considered because they are directly specified by the user.

An important note to make here, is that the choice of `USE` and `DEF` sets does not affect the soundness of the algorithm, but can affect the level of precision. The particular choice described above is based on the intuition that treats predicates as the variables of a program. Indeed, the abstraction predicates that are used in TVLA specifications are often derived from the variables of the program.

Example 4.4.1 Consider the running example whose CFG is shown in Fig. 4.4. Applying the the first analysis phase gives the result shown in Table 4.3.

The result indicates that no predicate is abstract at the exit node. This is trivial, since the exit node has no outgoing edges, and thus the predicates values of structures that are stored at that CFG node are never used in the analysis.

The result also indicates that the predicates y, $r[n, y]$ and t are not abstract at CFG node n_1. This is because the value of the variable `y` is modified by the statement `y = NULL` and the value of the variable `t` is modified at the beginning of the loop by the statement `t = y`. More significant is the fact that the predicate t is not abstract at CFG nodes n_2 and n_3 in the loop, and that the predicate y is not abstract at CFG node n_4 in the loop.

[1]Since, actions correspond to CFG edges, the sets are modified appropriately to obtain the `USE` and `DEF` sets corresponding to CFG nodes.

[2]Used for modelling conditions.

[3]Used to report textual messages to users.

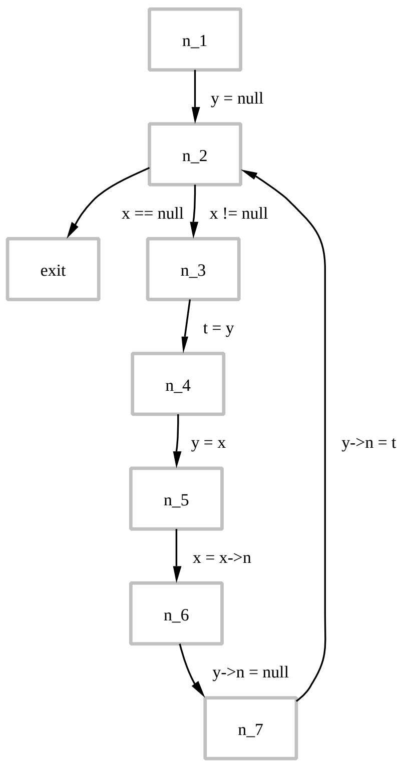


Figure 4.4: CFG of the running example.

In this example, our analysis determined that the predicates t and y are not abstract at exactly the cases where a live-variables analysis would determine that the corresponding program variables `t` and `y` are not alive (where their values don't affect the semantics of the program), which explains why the second phase loses no precision.

Interestingly, our analysis did not remove the predicates $r[n,t]$ and $r[n,y]$ from the subsets inside the loop, where the predicates t and y were removed. The reason is the abstract semantics of the statement `y->n = NULL`, which reads and updates the values of all predicates that are used for tracking reachability from program variables. This is unfortunate, since removing them from the appropriate subsets will not cause a loss of precision due to the fact that their precision is only important when the corresponding program variables are alive.

We applied our staged analysis in conjunction with the canonical names method to the benchmarks. The results are shown in Table 4.4.

The results show no improvement for the `MA` benchmark. This is not surprising, since its CFG contains only three nodes, of which only two nodes store structures, and our liveness analysis determines that the abstraction predicates at these nodes consist of all abstraction predicates in the specification. On other benchmarks the improvement is more notable. For example, on the last three benchmarks the number of structures was reduced by a factor of 1.49 for `insert-sort`, 1.96 for `bubble-sort` and 1.86 for `GC`, with similar effect on time and space performance.

Table 4.3: Result of applying the "predicate-liveness" analysis to the running example. The entire set of abstraction predicates that appear in the specification is $\{c[n]$, $is[n]$, $r[n,x]$, $r[n,y]$, $r[n,t]$, x, y, $t\}$.

CFG Node	**Live Abstraction Predicates**
n_1	$c[n], is[n], r[n,x], r[n,t], x$
n_2	$c[n], is[n], r[n,x], r[n,t], x, r[n,y], y$
n_3	$c[n], is[n], r[n,x], r[n,t], x, r[n,y], y$
n_4	$c[n], is[n], r[n,x], r[n,t], x, r[n,y], t$
n_5	$c[n], is[n], r[n,x], r[n,t], x, r[n,y], y, t$
n_6	$c[n], is[n], r[n,x], r[n,t], x, r[n,y], y, t$
n_7	$c[n], is[n], r[n,x], r[n,t], x, r[n,y], y, t$
exit	-

Table 4.4: Comparison of the results obtained with the canonical names method and the canonical names method with the liveness optimization. #strucs indicates the number of structures explored by the analysis, the time is measured in seconds and space is measured in Mb.

Benchmark	**Canonical Names Method**			**Canonical Names+Liveness**		
	#strucs	**time**	**space**	**#strucs**	**time**	**space**
`MA`	327	178	1.4	327	185	1.4
`reverse`	70	1	0	63	1	0
`merge-sorted`	319	6	1	315	6	0.6
`insert-sort`	934	18	2.1	628	14	1.8
`bubble-sort`	792	15	1.9	404	7	1
`GC`	1,128	9	1.9	605	5	0.8

4.5 An Even More Compact Domain Based on Pseudo-Embedding

Definition 4.5.1 *Let S and S' be two structures. Let $f : U^S \rightarrow U^{S'}$ be surjective. We say that f* ***pseudo-embeds*** *S in S' (denoted by $S \widetilde{\sqsubseteq}^f S'$) if for every unary* ***abstraction predicate*** *p and all $u \in U^S$,*

$$p^S(u) \sqsubseteq p^{S'}(f(u))$$

and for all $u' \in U^{S'}$

$$(|\{u | f(u) = u'\}| > 1) \sqsubseteq sm^{S'}(u')$$

We say that S ***can be pseudo-embedded in*** *S' (denoted by $S \widetilde{\sqsubseteq} S'$) if there exists a function f such that $S \widetilde{\sqsubseteq}^f S'$.*

We use pseudo-embedding as a compatibility condition for merging two structures S and S' when $S \widetilde{\sqsubseteq} S'$. This condition is more liberal than the condition used by the canonical names method. In particular it allows us to merge two structures with node sets of different size (recall that the canonical names method only merges structures that have the same set of canonical names.)

Unlike embedding, pseudo-embedding can be efficiently determined for any pair of structures, since all abstraction predicates have arity $k < 2$. Merging two structures is done by finding the pseudo-embedding function and joining predicate values accordingly.

We further modify the *blur* operation in the same spirit, by merging any two nodes u and v, such that for every unary abstraction predicate p the condition $p(u) \sqsubseteq p(v)$ holds (the existing *blur* operation demanded that $p(u) = p(v)$ holds.) Merging two nodes includes joining the values of the non-abstraction predicates and setting the value of the sm predicate to $1/2$ (i.e., the resulting node is a summary node).

Although, this is a theoretically more compact domain than the domain obtained from using the canonical names method, in practice we discovered that, for all of the benchmarks, it produced identical results. The reason is that our benchmarks produce only definite structures, and under this condition the method degenerates to the canonical names method (i.e., it is not possible to merge more nodes than are merged by the existing *blur* operation and it is not possible to merge more structures).

In order to achieve more compaction we turn to look for techniques that can generate non-definite structures. One such technique is the one presented in the previous section, which allows some predicates that were declared as abstraction predicates to become non-abstraction in parts of the CFG[4].

Applying the new compaction technique in conjunction with the liveness analysis technique we obtained the results shown in Table 4.5.

[4] In fact, this was one of the motivations for developing the technique of the previous section.

Table 4.5: Comparison of the results obtained with the canonical names method with the liveness optimization and the results obtained with the pseudo-embedding method with the liveness optimization. #strucs indicates the number of structures explored by the analysis, the time is measured in seconds and space is measured in Mb.

Benchmark	Canonical Names+Liveness			Pseudo-Embedding Method		
	#strucs	time	space	#strucs	time	space
`MA`	327	185	1.4	327	185	1.4
`reverse`	63	1	0	63	1	0
`merge-sorted`	315	6	0.6	314	6	0.8
`insert-sort`	628	14	1.8	362	5	1
`bubble-sort`	404	7	1	401	6	1.2
`GC`	605	5	0.8	413	3	0.7

The results show an improvement mainly for the `insert-sort` benchmark, where the number of structures was reduced by a factor of 1.7, and for the `GC` benchmark, where the number of structures was reduced by a factor of 1.5.

4.6 Comparing the New Domains to the Existing Domains

In this section, we conduct a comparison of results between the existing methods — relational and single-structure, and the pseudo-embedding (with liveness) method. Table 4.6 shows the results obtained with the three abstraction methods.

The results obtained for the `reverse` benchmark show very little improvement, which consumes very little resources with any method.

For the `MA` benchmark we see that the new method vastly improves over the relational method, which does not terminate in reasonable time. The single-structure does even better than the new method (about 4.5 times faster), but we note that this required manual labor to design nullary predicates to get the required precision with this method [NNS00].

On the rest of the benchmarks, the results indicate that the new method provides significant improvement relative to the relational method by requiring just a fraction of the resources. For example, the ratio between the number of structures required for the two methods is between 4 (for `merge-sorted`) and 460 (for `GC`), the time speedup is between 4.1 (for `merge-sorted`) and 1077 (for `GC`) and memory consumption is reduced between a factor of 5.1 (for `merge-sorted`) and 591 (for `GC`). It is important to note that the new method preserves the overall precision level as the relational method.

In contrast, we see that the single-structure method can sometimes be more costly than the relational method, such as in the case of the sorting benchmarks where it runs out of memory. For the other benchmarks, the method performs very well in terms of resources, but the analysis

Table 4.6: The results obtained for three abstraction methods. #strucs indicates the number of structures explored by the analysis, the time is measured in seconds and space is measured in Mb (mo indicates that the analysis ran out of memory). The notation a/b that appears rows for the single-structure method indicates how many structures were explored (the first number) and how many of them caused false alarms (second number). The other methods did not produce any false alarm.

Benchmark	**Relational**			**Single-Structure**			**Pseudo-Embedding**		
	#strucs	**time**	**space**	**#strucs**	**time**	**space**	**#strucs**	**time**	**space**
`MA`	>38,784	>14,400	>12.8	72	52	0	327	185	1.4
`reverse`	70	1	0	35 / 4	1	0	63	1	0
`merge-sorted`	1240	25	4.1	110 / 35	5	0	314	6	0.8
`insert-sort`	2,828	57	9.9	mo	mo	mo	362	5	1
`bubble-sort`	3,012	58	9.9	mo	mo	mo	401	6	1.2
`GC`	189,772	3,231	414	53 / 9	1	0	413	3	0.7

is overly imprecise (indicated by the false alarms).

Although both methods — single-structure and the new method (pseudo-embedding with liveness) attempt to improve relative to the relational method, there are a few differences between them that are worth noting:

- The asymptotic complexity of the single-structure method is exponential in the worst case (without nullary predicates), whereas the new method has the same worst case complexity as the relational method — doubly-exponential in the number of abstraction predicates.
- When the single-structure method merges a structure $S^1 = \langle U^1, \iota^1 \rangle$ with $S^2 = \langle U^2, \iota^2 \rangle$, the size of the node set of the resulting structure $S^3 = \langle U^3, \iota^3 \rangle$ is:

  $$\max\{|U^1|, |U^2|\} \leq |U^3| \leq |U^1| + |U^2|$$

  For the new method, the size is:

  $$|U^3| = \min\{|U^1|, |U^2|\}$$

- When the single-structure method merges two structure with different node sets, some of the nodes become maybe-active. This sometimes prevents the analysis from detecting (by use of TVLA's constraint-solver) structures that represent impossible configurations, where a more precise analysis would detect these situations and remove them (causing a memory leakage false-alarm on the running example). This can add to the growth in the size of the structures, as mentioned in the previous item, which leads to memory explosion. The new method never introduces maybe-active nodes, and avoids this problem.

- The single-structure method sometimes propagates large structures even though only small portions of them change between consecutive iterations. On the other hand, the set-based methods (relational and new method) use an incremental algorithm that propagates only those structures that caused a change to the set of structures. This can lead to a more efficient analysis.

Chapter 5

Conclusion

In this thesis, we attempted to attack the scalability issue of shape analysis in the TVLA system.

In the first part, we dealt with the memory consumption problem by designing two new data structures. Both of the data structures employed similar techniques for sharing information, which resulted in space reduction by a factor of 4 to 10 without compromising time performance. Our experience has been that these representations also help reduce the time required for software verification, though we have not addressed that aspect in detail in this thesis.

From the empirical study we are able to conclude the following:

- The most significant contribution to the space reduction is due to the normalization techniques.
- The amount of sharing obtained by canonical representation of substates is significantly more than the one obtained by using inherited sharing — the normalization techniques allow to more successfully capture invariants that are common to different program states.
- While OBDDs are based on the principles of propositional logic, the techniques employed by the Functional representation are data structure oriented. However, both of the representations seem to produce results of similar quality. This indicates, that normalization via hash-consing, which is common to both representations, is the most important principle.
- It is worth noting that the empirical results presented in this thesis show that our representations use just a few (at most 3) objects per structure per program point, on the average. This implies that the representations are fairly efficient and that room for further savings exists only if we move on to extensions such as the representation of sets of structures.

TVLA's notion of program state is quite general and expressive, and can be used to encode a wide variety of program analysis algorithms—particularly those that are flow- or context-sensitive, and represent states as maps or directed graphs. We therefore believe that our results are also likely to be applicable to other static analysis and verification systems.

In the second part of the thesis, we attempted to reduce the overall work required for analyses by exploring different abstractions. We were able to find several techniques that produced significant improvement to the space and time performance, sometimes by several orders of magnitude, without sacrificing the precision of the analyses.

In addition, the technique we used in Section 4.4 shows that *staged analysis*, which is an approach used by other static analysis algorithms, can be useful in the context of shape analysis as well.

5.1 Related Work

Data Structures for Static Analysis Several recent and ongoing research efforts have explored the use of OBDDs in the context of static analysis, but mostly for domains simpler than first-order structures. PAG [Mar98] and SLAM [Mic01] use OBDDs to represent sets ("bit vectors") and interpretations of *propositional* (rather than first-order) structures. PAG also employs persistent data structures for static analysis, exploiting inherited sharing. Mona uses BDDs to represent transitions of a tree automaton [KMS00], which is used to implement a decision procedure used for Hoare-style verification. Mauborgne [Mau98] explores the use of TDGs (a refinement of OBDDs) in abstract interpretation, using them to encode higher-order functions for strictness analysis, and presents empirical results on analysis time (but not on memory usage). It is not obvious from this prior work how OBDDs can be used beneficially for sophisticated analyses, such as heap shape analysis, that use domains based on first-order structures.

Using OBDDs to represent first order logical structures Veith [Vei98] describes a representation of a logical structure, where each predicate interpretation is encoded by a separate OBDD, and a vector of predicate interpretations is used to represent a structure. We achieve better sharing by encoding a complete structure using a single OBDD.

Other first-order state representations The composite symbolic library [YKTB01] can model a limited form of first-order state using formulae of Presburger arithmetic.

Staged Analysis Finally, we note that the analysis engine of ESP [ABD$^+$] also uses a staged analysis approach. The engine combines two separate phases of program analysis: Phase one is a global context-sensitive control-flow-insensitive analysis that produces a call graph and information about the flow of values in the program. Phase two is an inter-procedural context-sensitive dataflow analysis that incorporates a restricted form of inter-procedural path simulation. The first phase produces the program abstractions used by the second phase. In contrast, to ESP, which uses the first to refine the abstraction, our staged analysis uses the first phase to find opportunities for coarsening the abstraction without losing too much precision.

5.2 Further Work

The results reported in this thesis show a significant improvement in the performance of shape analyses in TVLA. However, the goal of creating scalable shape analyses is still not achievable with the current techniques. The experience we gained suggests several more opportunities for improving performance.

Compactly representing sets of first-Order structures In our work, we concentrated on developing data structures for compactly representing structures. The experiments indicate that the representations are fairly efficient and that room for further savings exists only if we move on to extensions such as the representation of sets of structures. Opportunities for sharing between sets of structures occur, for example, when program conditions are modelled, since conditions have the effect of passing a subset of the structures to each of the target CFG nodes.

Increasing the locality of abstractions In section Section 4.4 we noted that the current technique misses opportunities for localizing abstraction for predicates that track reachability from program variables. We observe that a better technique can infer that these predicates depend on other predicates, which correspond to program variables, and should therefore “inherit” their liveness properties.

Symbolic execution of TVLA operations Currently, one of the major bottle-necks is that of handling single program statements. The complexity can be high due to the cost of evaluating formulae in TVLA and the fact that many predicates can be updated by a single statement. We have begun to consider symbolic execution techniques for reducing the costs of handling program statements. By breaking the problem of handling a single statement into smaller sub-problems, the symbolic techniques can identify equivalent sub-problems and avoid repeated computations.

Handling real-life programs In addition to the work discussed in the previous chapters, we also dedicated effort in creating software infrastructure that will help us to achieve the goal of obtaining scalable shape analyses. Analyzing programs with TVLA requires the CFG of the program to be supplied in the TVP language. Although this is a generic format, translating from a high-level language, such as Java, to TVP is a tedious and error-prone task, which prohibits analyzing but very small programs. For this reason, we created the J2TVLA toolkit that aims at automating this process for Java programs, by supplying a set of tools that make this task feasible.

References

[ABD$^+$] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. 2477:230–246.

[AG98] A.W. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, New York, Cambridge, 1998.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *Design Automation Conference*, pages 40–45, June 1990.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986. Electronic version with annotations available as http://www.cs.cmu.edu/∼bryant/pubdir/ieeetc86.ps.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 269–282, New York, NY, 1979. ACM Press.

[CG98] L. Cardelli and A.D. Gordon. Mobile ambients. In M. Nivat, editor, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, March 1998.

[DRS00] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Checking cleanness in linked lists. In *Proc. Static Analysis Symp.*, pages 115–134, July 2000.

[KMS00] Nils Klarlund, Anders Moller, and Michael I. Schwartzbach. MONA implementation secrets. In *CIAA*, pages 182–194, 2000.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In Jens Palsberg, editor, *Proc. Static Analysis Symp.*, volume 1824 of

Lecture Notes in Computer Science, pages 280–301. Springer-Verlag, 2000. http://www.cs.tau.ac.il/∼ rumster/TVLA/.

[Mar98] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[Mau98] Laurent Mauborgne. Abstract interpretation using typed decision graphs. *Science of Computer Programming*, 31(1):91–112, May 1998.

[Mic01] Microsoft Research. The SLAM project. http://research.microsoft.com/slam/, 2001.

[NNS00] F. Nielson, H.R. Nielson, and M. Sagiv. A Kleene analysis of mobile ambients. In G. Smolka, editor, *Proc. of ESOP 2000*, volume 1782 of *LNCS*, pages 305–319. Springer, 2000.

[RSW01] T. Reps, M. Sagiv, and R. Wilhelm. Automatic verification of a simple mark and sweep garbabge collector. Presented in the 2001 University of Washington and Microsoft Research Summer Institute, Specifying and Checking Properties of Software, http://research.microsoft.com/specncheck/, 2001.

[RWF$^+$02] G. Ramalingam, Alex Warshavsky, John Field, Deepak Goyal, and Mooly Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2002. (To appear.).

[RWFS01] G. Ramalingam, Alex Warshavsky, John Field, and Mooly Sagiv. Deriving specialized heap analyses for verifying component-client conformance. Technical Report RC22145, IBM T.J. Watson Research Center, 2001. http://domino.watson.ibm.com/library/cyberdig.nsf/papers?SearchView&Query=RC22145.

[Som98] F. Somenzi. Colorado University Decision Diagram Package (CUDD). Department of Electrical and Computer Engineering, University of Colorado at Boulder, 1998. http://vlsi.colorado.edu/∼ fabio/CUDD/.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[Vei98] H. Veith. How to encode a logical structure by an obdd. In *IEEE Conference on Computational Complexity*, pages 122–131, 1998.

[Yah01] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 27–40, 2001.

[YBO+98] Bwolen Yang, Randal E. Bryant, David R. O'Hallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev K. Ranjan, and Fabio Somenzi. A performance study of BDD-based model checking. In *Proceedings of the Formal Methods on Computer-Aided Design*, pages 255–289, November 1998.

[YKTB01] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. Composite symbolic library. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 335–344. Springer-Verlag, April 2001.

List of Figures

List of Tables

Appendix A

Normalization of the Functional Representation

We now explain how a structure's functional representation is normalized.

(a) First, the fliks used in a structure are normalized. This includes the flik representing the nullary predicate values, as well as the fliks occurring as second level maps in the representation of unary and binary predicate values. Note that since the canonical name of an individual (i.e., the value of abstraction unary predicates for that individual) is also represented by a flik, this creates unique representatives for the canonical name of individuals. Subsequent to this step, canonical names can be compared for equality via a pointer equality comparison. During this process, the unique representatives of canonical names are each assigned an unique integer value. This also establishes an arbitrary, but fixed, total order on canonical names.

(b) The linked list of individuals (representing the universe) is then sorted with respect to this total order on canonical names of individuals.

(c) The individuals are then *renumbered* from 0 to $s - 1$, where s is the number of individuals. The intmaps and intpairmaps representing the values of unary and binary predicates are simultaneously regenerated to account for the individual renumbering. The regenerations of these maps utilizes hash-consing to generate canonical representatives of these maps as well. A canonical representative for the linked list representing the universe is also obtained trivially from the cardinality of the universe.

If $\langle p_1, p_2, p_3, p_4, p_4 \rangle$ and $\langle q_1, q_2, q_3, q_4, q_5 \rangle$ represent two structures completely normalized as above, these structures are isomorphic iff every p_i equal to q_i (pointer comparison).

Appendix B

Signature of ADT for Evolving First-Order Structure

The ADT for evolving first-order logical structures is shown in Fig. B.1.

```
type Kleene /* 0, 1, or 1/2 */
type Node /* individual in structure */
type NullaryPredicate
type UnaryPredicate
type BinaryPredicate
type TVS /* mutable 3-valued structure */
type ImmutableTVS /* immutable 3-valued structure */
type TVS_SET /* set of 3-valued structures */

/* evaluate predicate value for specified node(s) in TVS */
eval: TVS * NullaryPredicate -> Kleene
eval: TVS * UnaryPredicate * Node -> Kleene
eval: TVS * BinaryPredicate * Node * Node -> Kleene

/* update predicate to specified Kleene value for specified node(s) in TVS */
update: TVS * NullaryPredicate * Kleene -> void
update: TVS * UnaryPredicate * Node * Kleene -> void
update: TVS * BinaryPredicate * Node * Node * Kleene -> void

empty_TVS: TVS /* a TVS with empy universe */
empty_SET: void -> TVS_SET /* returns a new, empty, set */
copy: TVS -> TVS /* copy mutable TVS */
immutableCopy: TVS -> ImmutableTVS /* generate immutable copy of mutable TVS */
universe: TVS -> (Set of Node) /* enumerate nodes in TVS's universe */
new: TVS -> Node /* add a node to the TVS's universe */
remove: TVS * Node -> void /*remove node from TVS */
add: TVS_SET * ImmutableTVS -> bool /* add TVS to set */
```

Figure B.1: ADT for evolving first-order structures.

Appendix C

Explanation of Abstract Counters for Different Representations

Let n denote the number of nodes in a TVS and let $|NP|$, $|UP|$, and $|BP|$ denote the number of nullary, unary and binary predicates respectively. We compare the space required by the three representations with two different measures of the size of the structures produced during the analysis. The first of these metrics, the *dense* metric, corresponds to the amount the space required to store every predicate value explicitly. The space required for the dense representation is $(|NP|$ + $n\ |UP|$ + $n^2\ |BP|)/4$ bytes (assuming that 4 kleene values are stored in each byte). The second metric is the *sparse* metric in which only the non-zero predicate values are stored. Although, the actual number of values stored for the sparse representation is smaller than that for the dense representation, the overhead required to store each individual value is much higher as one must store the nullary predicate, or the unary predicate and node, or the binary predicate and pair of nodes with the kleene value. Assuming that each value can be packed into a single word (4 bytes), the physical space required for the sparse representation is 4 times the number of non-zero predicate values.

The Base implementation uses exploits both sparseness and sharing to a limited extent. Let NP_1, UP_1, and BP_1 denote the number of non-zero nullary, unary, and binary predicate entries stored in the first-level table. Let n_1 denote the non-zero entries in the second-level tables for unary predicates. Note that n_1 must be greater than 0 (otherwise, the unary predicate is zero for all nodes, and should not have an entry in the first-level table) and can be n in the worst-case. Let n_2 denote the non-zero entries in the second-level tables for binary predicates. Since each such entry corresponds to a pair of nodes, n_2 can be n^2 in the worst case. Let $unique$ denote a conditional value that is used to count only unique objects. Thus, for each shared object, $unique$ is 1 for only one occurrence of the object and 0 for all other occurrences. Then, the space required for each TVLA structure in the Base representation can be computed as

$$5 + 4 \times \mathit{unique} \times n + 8 \times (NP_1 + UP_1 + BP_1) + 5 \times (UP_1 + BP_1) + \\ 5 \times \mathit{unique} \times n_1 + 9 \times \mathit{unique} \times n_2.$$

The node set requires 4 bytes for the pointer to the possibly shared node set and 1 byte for the shared bit. If the node set is not shared then 4 bytes are required for each node. For all entries in the first-level table 8 bytes are needed for the predicate id and a pointer to its bindings. For unary and binary predicates another level of indirection, consisting of a pointer and a shared flag take another 5 bytes for each such entry in the first-level table. In the second-level table, 5 bytes are needed for (node,kleene) pairs and 9 bytes are needed for (node,node,kleene) triples.

For the OBDD implementation, the CUDD package provides information about the maximum number of live OBDD nodes that existed during the analysis. We multiply this number by 20, assuming that an OBDD node can be implemented using 20 bytes, to get the space statistic for the OBDD implementation,

The number reported for the Functional implementation is obtained by multiplying the number of objects used by the representation by 24 (as all objects in this implementation require 24 bytes or less).

Appendix D

Proving the Hardness of Embedding

Lemma D.0.1 *GRAPH ISOMORPHISM $\leq_\rho$ STRUCTURE EMBEDDING*

Proof: Notice that first-order logical structures generalize directed graphs, since the edge relation can be encoded with a suitable binary predicate — $edge$. To prove the claim, we use the following reduction. Given a pair of possibly isomorphic graphs $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$, we pass the same input to the embedding problem — the graphs are viewed as structures where the $edge^{G_1}(u, v) \Leftrightarrow (u, v) \in E_1$ and $edge^{G_2}(u, v) \Leftrightarrow (u, v) \in E_2$.

$\Longrightarrow$ Suppose that G_1 and G_2 are isomorphic. Then there exists a one-to-one function $f : V_1 \to V_2$, such that for all $u, v \in V_1$ we have $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$. Therefore, $edge(u, v)^{G_1} \sqsubseteq edge(f(u), f(v))^{G_2}$ and $G_1 \sqsubseteq^f G_2$.

$\Longleftarrow$ Suppose that $G_1 \sqsubseteq^f G_2$. Then for the predicate $edge$ we have $edge(u, v)^{G_1} \sqsubseteq edge(f(u), f(v))^{G_2}$, but since for all $u, v \in U^{G_1}$ and for all $u, v \in U^{G_2}$, it is true that $edge(u, v) \in \{0, 1\}$ (a graph edge either exists or doesn't exist) and therefore $edge(u, v)^{G_1} \Leftrightarrow edge(f(u), f(v))^{G_2}$. From the definition of embedding we also know that f is surjective. In addition, for all $u \in G_2$ it is true that $sm(u) \in \{0, 1\}$ (a graph vertex represents only itself) and therefore f is one-to-one.