

Compactly Representing First-Order Structures for Static Analysis

R. Manevich^{1,*}, G. Ramalingam², J. Field², D. Goyal², and M. Sagiv¹

¹ Tel Aviv University {rumster, msagiv}@post.tau.ac.il

² IBM T.J. Watson Research Center {rama, jfield, dgoyal}@watson.ibm.com

Abstract. A fundamental bottleneck in applying sophisticated static analyses to large programs is the space consumed by abstract program states. This is particularly true when analyzing programs that make extensive use of heap-allocated data. The TVLA (Three-Valued Logic Analysis) program analysis and verification system models dynamic allocation precisely by representing program states as *first-order structures*. In such a representation, a finite collection of *predicates* is used to define states; the predicates range over a universe of individuals that may evolve—expand and contract—during analysis. Evolving first-order structures can be used to encode a wide variety of analyses, including most analyses whose abstract states are represented by directed graphs or maps. This paper addresses the problem of space consumption in such analyses by describing and evaluating two novel structure representation techniques. One technique uses *ordered binary decision diagrams* (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. In addition to TVLA, we believe that our results are applicable to many program analysis systems that represent states as graphs, maps, or other structures of similar complexity.

1 Overview and Main Results

A fundamental bottleneck in applying sophisticated static analyses to large programs is the space consumed during analysis by abstract program states. This is particularly true for modern programs where much of the data manipulated by the application lies in the heap (i.e., in dynamically allocated data structures), instead of being stored in a fixed set of variables. An abstraction of the potentially unbounded heap that is precise enough for a particular analysis may often have prohibitive space requirements. The TVLA (Three-Valued Logic Analysis) program analysis and verification system [6] is designed to model dynamic allocation precisely by representing program states as *first-order structures*. In such a representation, a finite collection of *predicates* is used 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.,

* Partially supported by the Israeli Academy of Science. Part of this work was done while visiting IBM’s T.J. Watson Research Center.

dynamic memory allocation or dynamic thread creation to be modelled in a natural way [13, 17]. In addition, first-order state representations can be much more compact than representations based on finite-state models. For example, when representing programs with pointers, a natural finite state representation of pointer aliases usually consumes quadratic space [14], whereas a graph based (first-order) representation such the one used by TVLA can be linear in many cases.

TVLA has been successfully applied to a wide variety of deep program analysis and verification problems, including checking C programs for the presence of memory leaks and dangling or uninitialized pointers [4], verifying the correctness of a simple garbage collector [12], determining whether clients of a Java library satisfy the library’s *conformance constraints* for correct usage [11], and verifying certain safety properties of a packet router [10]. In carrying out these tasks, TVLA utilizes an abstract interpretation that maintains detailed information on the state of dynamically allocated memory. While this degree of precision is crucial to avoid reporting an excessive number of “false alarms”, it comes at a price: the space required to analyze programs of more than a few thousand lines of code is often prohibitive.

This paper 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* [2, 18] (OBDD) implementation [15] 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*. Both of these core data structures are well-known, and OBDDs have been used in several program analysis systems to represent automata or finite sets. However, it is not obvious that they should be adaptable to, or beneficial for, representing evolving first-order structures, especially given their poor worst-case behavior.

Happily, 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. Since the notion of evolving first-order structure is powerful enough to encode a number of nontrivial state representations (e.g., those based on directed graphs or maps), we believe that our results are also likely to be applicable in a wide variety of program analysis techniques requiring such structures.

2 TVLA Primer

In this section, we give a brief overview of those aspects of the TVLA system that pertain to state manipulation. Complete details of the system may be found in [6].

2.1 Program States as Three-Valued Logical Structures

Given a collection of finite-arity predicates, a *first-order logical structure* is a collection of *individuals* (nodes), along with an *interpretation* (truth value assignment) of each predicate for every tuple of individuals in the predicate’s domain. In order to bound

the number of distinct first-order structures generated during analysis and thus ensure termination, TVLA must sometimes *blur* a structure by merging several individuals together. Blurring typically yields a *three-valued* logical structure, in which the third value, $1/2$, denotes facts which may have been either 0 (false) or 1 (true) in the state prior to blurring. Logical formulas used to update or query the state are then interpreted appropriately over a three-valued structure. In general, each 3-valued logical structure represents a (possibly infinite) *set* of 2-valued structures. Nodes in a 3-valued structure that may represent more than one individual from a given 2-valued structure are called *summary nodes*. The set of individuals comprising a 3-valued structure S is called its universe (denoted by U^S); $p^S : (U^S)^k \rightarrow \{0, 1, 1/2\}$ denotes the interpretation of predicate p of arity k in structure S . (When depicting an interpretation function, we will usually omit the superscript denoting a structure when it is evident from context.)

Consider the program depicted in Fig. 1, which prints all the elements of a singly-linked list. As an expository example, 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* [13] can be used for this purpose. The predicates used to define the shape analysis operate on individuals representing list elements. Pointer variables will be represented by unary predicates; for example, $y^S(u) = 1$ if the variable y points to the list element represented by u in structure S . Pointer *fields* within the list elements will be represented as binary predicates, thus $n^S(u_1, u_2) = 1$ if the n -field of u_1 points to u_2 in structure S . (Although the first-order machinery of TVLA is not really necessary for this simple example, it is crucial for more complicated examples, e.g., for proving that a dereferenced pointer *field* is not `NULL`.)

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

Fig. 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 .

In TVLA, predicates are required to have arity ≤ 2 ; hence it is natural to depict structures in the form of directed graphs. Consider the 3-valued structure S_0 depicted in Fig. 3. Here, the variable y is represented by a unary predicate y , which has value 1 only for u_1 . In general, a unary predicate p is represented graphically by a solid arrow connecting p to each individual u for which $p(u) = 1$, a dotted arrow if $p(u) = 1/2$, 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 x indicates that variable x is null in S_0 . A

solid directed edge labeled by p from u_1 to u_2 denotes the fact that $p(u_1, u_2) = 1$; a dotted edge denotes a $1/2$ value. TVLA uses a special unary predicate sm to maintain summary node information: $sm(w) = 1/2$ for a summary node w if w may represent multiple nodes in a corresponding 2-valued structure; otherwise, $sm(w) = 0$ if w represents a unique node. Given these conventions, we see that S_0 may be thought of as representing an infinite number of concrete data structures, including all lists of length 2 or more pointed to by program variable γ , as well as lists with cyclic and shared tails, or lists with collections of unreachable (“garbage”) elements.

```

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 (ActionSequence( $P, P'$ ),  $S'$ )
        push(stack, [ $P', S'$ ])
      }
    }
  }
}

```

Fig. 2. The structure of TVLA’s abstract interpretation algorithm.

2.2 Abstract Interpretation in TVLA

TVLA takes as input a control flow graph (CFG) of the program to be analyzed—each edge of which is 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), 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 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 $x = x \rightarrow n$ in Fig. 1 is modelled 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 exploring all abstract program states derivable from the initial set of states via action sequences associated with CFG edges, using the algorithm outlined in Fig. 2. The actions which may be associated with each CFG edge are enumerated in Fig. 4, and described in greater detail in Section 3. The action sequences 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. (The blur operation merges nodes that have the same *canonic* name together, where a node’s canonic name is defined to be the sequence of values it has for the unary predicates. This guarantees that a blurred structure can have at most one node with a given canonic name.) Termination of the algorithm is guaranteed by the fact that there are only finitely many blurred structures.

The structures generated by abstract interpretation of the first iteration of the loop body of the `print_all` function are depicted in Fig. 3. The analysis begins with the 3-valued structure S_0 . The actions modelling the statement $x=y$ have the effect of setting x to point to u_1 , resulting in the structure S_1 . The most interesting case is the assignment $x = x \rightarrow n$. This statement is modeled by the predicate update action for $x = x \rightarrow n$ 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 incoming structures, which frequently sharpens the analysis results. Here, for example, focus allows us to distinguish the case where $x \rightarrow n$ is null in the input structure, producing an output structure where x is null (represented by structure $S_{2.0}$) from the input structures where $x \rightarrow n$ is non-null.

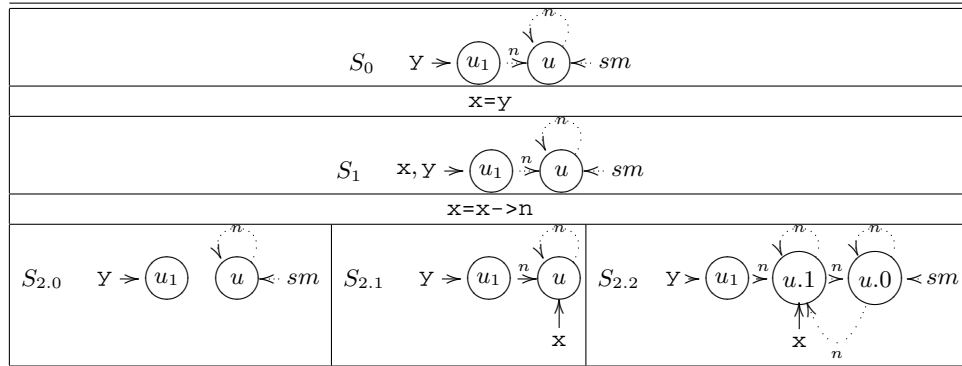


Fig. 3. Structures generated by abstract interpretation of the first iteration of the loop in the `print_all` function.

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.

3 Evolving First-Order Structures as an Abstract Data Type

In the remainder of the paper, we will describe and evaluate space-efficient representations for evolving first-order structures. These representations exploit various opportuni-

ties for space savings, most of which arise in conjunction with a variety of abstract state representations (including, but not limited to, first-order structures). These opportunities include:

- *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. 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. One reason why equivalent substructures often arise is that each program point typically has a set of invariants that hold true for all states associated with that point. Such invariant properties often give rise to equivalent substructures, which can in turn produce significant opportunities for space saving. This is particularly true for program points inside loops, where differing states will often contain loop-invariant substructures.
- *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.

Before describing the different representations we study, we first present the full signature of the abstract data type (ADT) used to represent evolving first-order structures in TVLA (see Fig. 4). This signature describes the set of operations that any implementation must support.

Note that the ADT contains two distinct representations of structures: a mutable structure (TVS) and an immutable blurred structure (ImmutableTVS). In addition, it defines an interface to a set of structures (TVS_SET), which contains only immutable blurred 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

```

type Kleene /* 0, 1, or 1/2 */
type Node /* individual in structure */
type NullaryPredicate
type UnaryPredicate
type BinaryPredicate
type TVS /* mutable three-valued structure */
type ImmutableTVS /* immutable, blurred, three-valued structure */
type TVS_SET /* set of three-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 empty universe */
empty_SET: void -> TVS_SET /* returns a new, empty, set */
copy: TVS -> TVS /* copy mutable TVS */
immutableCopy: TVS -> ImmutableTVS /* returns a blurred, immutable copy */
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 */

```

Fig. 4. ADT for evolving first-order structures.

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 TVS Representations

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

4.1 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.

Note that this representation is not a completely naive representation: it exploits sparsity and a limited form of sharing, which yield significant space savings.

4.2 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. Due to lack of space, we will assume that readers are familiar with OBDDs, which may be used to represent boolean functions over a set of boolean variables.

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 node tuple 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 $|N_1| = |N_2| = \lceil \log |U^S| \rceil$ OBDD variables.

Fig. 5 shows the OBDD representation of the structures S_0 , S_1 and $S_{2.2}$ from the running example (see Fig. 3). The OBDD nodes labelled $u1$ or $\langle u, u.1 \rangle$, for example, represent corresponding node tuples. 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 0101 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 0101 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 $x = x \rightarrow n$ ($S_{2.0}$, $S_{2.1}$ 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.

4.3 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 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.

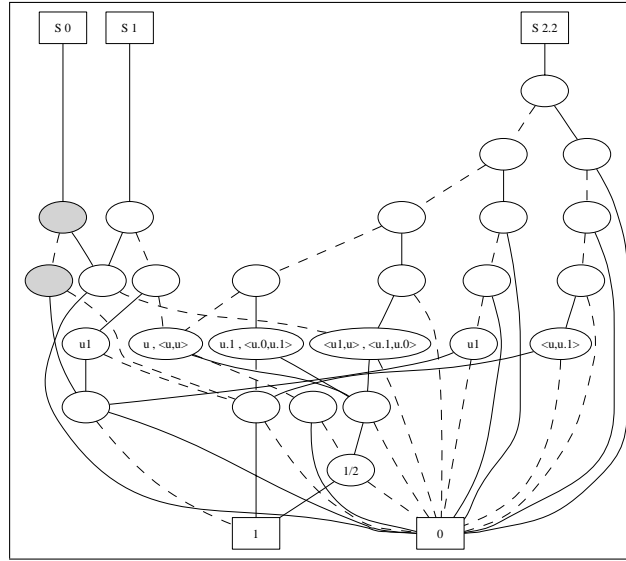


Fig. 5. The OBDDs representing the 3-valued structures in Fig. 3. Dotted edges denote 0, and solid edges denote 1. The node ordering imposed by the canonic names is $u < u_1$ for S_0 and S_1 , and $u.0 < u.1 < u_1$ for $S_{2.2}$. The predicates are numbered as follows: $sm : 0, x : 1, y : 2, n : 3$.

The values of nullary predicates in a structure are represented by a map from integers in the range $[0 : n_0]$ to Kleene. We use a tree-based functional data structure¹, 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

¹ 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.

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.)

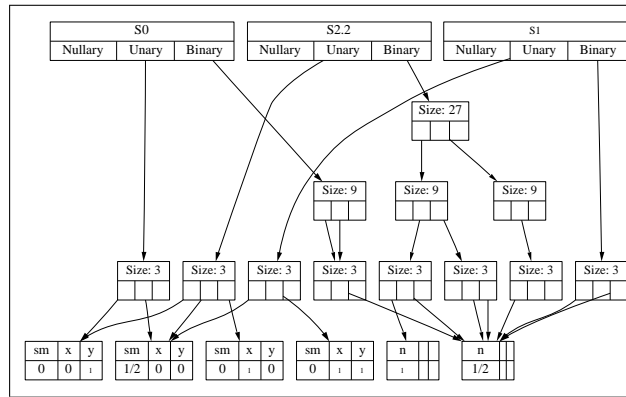


Fig. 6. The functional representation of the 3-valued structures in Fig. 3. Lists representing the universes are not shown in the figure. Nodes u_1 and u of structure S_0 are assigned numbers 0 and 1, respectively. Nodes u_1 and u of structure S_1 are, however, assigned numbers 1 and 0 respectively. 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.

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 *flick* (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 *flick*) that maps a binary predicate p 's number to the value of $p(m, n)$ in the structure.

Fig. 6 shows a representation of the structures S_0 , S_1 and $S_{2,2}$ from the running example. 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 flicks 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.

5 Empirical Evaluation

In this section we present an empirical evaluation of the representations described in Section 4. 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 [15] package.

Benchmarks The benchmarks used in the experiments are explained below. The `CA` benchmark performs “cleanness analysis” [4] in a C program implementing the instruction selection phase of the Tiger educational compiler [1]. The `GC` benchmark [12] 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 [11]. CMP requires identifying a specific type of misuse of Java Collection Classes. Finally, the `MA` benchmark [10] verifies certain safety properties of a packet router in the mobile ambient calculus [3].

Results Table 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.

Table 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

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 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 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,

Table 2. Space counters for different representations. These counters indicate the number of bytes required to represent the structures. 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

we computed the instrumentation-based space usage estimate periodically. Table 3 shows 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. Abstract counters used to represent TVS at different execution points of the iterative procedure. For MA we sample every $k = 2000$ structures, for CA $k = 5000$, and for GC $k = 10,000$. **D, S, B, O,** and **F** refer to the Dense, Sparse, Base, OBDD, and Functional representations respectively.

Sam- ple	MA					CA					GC				
	D	S	B	O	F	D	S	B	O	F	D	S	B	O	F
1	932	395	978	89	70	155	228	505	74	50	46	148	162	24	37
2	938	399	898	60	52	154	252	531	74	53	49	157	209	32	44
3	938	400	899	49	48	165	261	539	68	48	49	155	204	30	41
4	937	403	917	51	47	168	266	547	68	47	48	153	190	25	38
5	939	405	955	50	45	173	267	554	65	46	49	160	215	28	40
6	941	406	963	48	44	186	271	561	61	44	50	165	203	24	38
7	943	407	966	46	41	188	273	565	60	44	51	171	193	21	36
8	943	408	921	40	36	187	276	563	58	44	51	171	193	23	36
9	943	408	884	36	32	190	278	561	56	43	51	173	211	25	38
10	943	409	854	32	29	192	281	564	54	42	52	174	225	26	40

6 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 [7] and SLAM [9] 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 [5], which is used to implement a decision procedure used for Hoare-style verification. Mauborgne [8] 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 [16] 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 Finally, we note that the composite symbolic library [19] can model a limited form of first-order state using Presburger formulas.

7 Conclusion and Future Work

In this paper, we have looked at efficient data structures for representing first-order structures in the context of software verification and sophisticated static analyses. We have described two new representations for first-order structures and have empirically measured the effectiveness of these representations. Our results show that these representations are very promising and that they can greatly reduce the space requirements for software verification. 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 paper. We are currently working on more efficient algorithms for the various high-level operations, such as *coerce* and *focus*, used in verification via 3-valued logic and hope to empirically evaluate the time complexity of these algorithms with the new representations described in this paper.

The representations described in this paper are for single 3-valued structures. A relational analysis, however, computes a *set* of 3-valued structures at each program point. We are also working on extensions to these representations that can represent sets of 3-valued structures (e.g., the BDD representation of single structures can be extended to represent a set of structures). It is worth noting that the empirical results presented in this paper show that our representation uses just a few (at most 3) objects per structure per program point, on the average. This implies that the representation is fairly efficient and that room for further savings exist only if we move on to extensions such as the representation of *sets of structures*.

References

1. A. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, New York, Cambridge, 1998.

2. R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.
3. L. Cardelli and A. 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, Mar. 1998.
4. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Proc. Static Analysis Symp.*, pages 115–134, July 2000.
5. N. Klarlund, A. Moller, and M. I. Schwartzbach. MONA implementation secrets. In *CIAA*, pages 182–194, 2000.
6. T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In J. 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/>.
7. F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
8. L. Mauborgne. Abstract interpretation using typed decision graphs. *Science of Computer Programming*, 31(1):91–112, May 1998.
9. Microsoft Research. The SLAM project. <http://research.microsoft.com/slam/>, 2001.
10. F. Nielson, H. 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.
11. G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 83–94, Berlin, June 2002.
12. T. Reps, M. Sagiv, and R. Wilhelm. Automatic verification of a simple mark and sweep garbage 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.
13. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 105–118, 1999.
14. S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to data flow analysis problems. *Acta Inf.*, 35(6):457–504, June 1998.
15. 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/>.
16. H. Veith. How to encode a logical structure by an OBDD. In *IEEE Conference on Computational Complexity*, pages 122–131, 1998.
17. 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.
18. B. Yang, R. E. Bryant, D. R. O’Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. In *Proceedings of the Formal Methods on Computer-Aided Design*, pages 255–289, November 1998.
19. 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.