Symbolically Computing Most-Precise Abstract Operations for Shape Analysis[⋆]

G. Yorsh[1⋆⋆], T. Reps[2], and M. Sagiv[1]

[1] School of Comp. Sci., Tel-Aviv Univ., {gretay, msagiv}@post.tau.ac.il
[2] Comp. Sci. Dept., Univ. of Wisconsin, reps@cs.wisc.edu

Abstract. Shape analysis concerns the problem of determining "shape invariants" for programs that perform destructive updating on dynamically allocated storage. This paper presents a new algorithm that takes as input an abstract value (a 3-valued logical structure describing some set of concrete stores X) and a precondition p, and computes the most-precise abstract value for the stores in X that satisfy p. This algorithm solves several open problems in shape analysis: (i) computing the most-precise abstract value of a set of concrete stores specified by a logical formula; (ii) computing best transformers for atomic program statements and conditions; (iii) computing best transformers for loop-free code fragments (i.e., blocks of atomic program statements and conditions); (iv) performing interprocedural shape analysis using procedure specifications and assume-guarantee reasoning; and (v) computing the most-precise overapproximation of the meet of two abstract values.

The algorithm employs a decision procedure for the logic used to express properties of data structures. A decidable logic for expressing such properties is described in [5]. The algorithm can also be used with an undecidable logic and a theorem prover; termination can be assured by using standard techniques (e.g., having the theorem prover return a safe answer if a time-out threshold is exceeded) at the cost of losing the ability to guarantee that a most-precise result is obtained. A prototype has been implemented in TVLA, using the SPASS theorem prover.

1 Introduction

Shape-analysis algorithms (e.g., [11]) are capable of establishing that certain invariants hold for (imperative) programs that perform destructive updating on dynamically allocated storage. For example, they have been used to establish that a program preserves treeness properties, as well as that a program satisfies certain correctness criteria [8]. The TVLA system [8] automatically constructs shape-analysis algorithms from a description of the operational semantics of a given programming language, and the shape abstraction to be used.

The methodology of abstract interpretation has been used to show that the shape-analysis algorithms generated by TVLA are *sound* (conservative). Technically, for a given program, TVLA uses a finite set of abstract values L, which forms a join semi-lattice, and an adjoined pair of functions (α, γ), which form a Galois connection [2]. The abstraction function α maps potentially infinite sets of concrete stores to the *most-precise* abstract value in L. The concretization function γ maps an abstract value to the set of concrete stores that the abstract value represents. Thus, soundness means that the set of concrete stores

[⋆] Supported by ONR contract N00014-01-1-0796.
[⋆⋆] Supported in part by the THE ISRAEL SCIENCE FOUNDATION (grant No. 304/03)

$\gamma(a)$ represented by the abstract values a computed by TVLA includes all of the stores that could ever arise, but may also include superfluous stores (which may produce false alarms).

1.1 Main Results

The overall goal of our work is to improve the precision and scalability of TVLA by employing decision procedures. In [15], we show that the concretization of an abstract value can be expressed using a logical formula. Specifically, [15] gives an algorithm that converts an abstract value a into a formula $\widehat{\gamma}(a)$ that exactly characterizes $\gamma(a)$—i.e., the set of concrete stores that a represents.[3] This is used in this paper to develop algorithms for the following operations on shape abstractions:

- Computing the most-precise abstract value that represents the (potentially infinite) set of stores defined by a formula. We call this algorithm $\widehat{\alpha}(\varphi)$ because it is a constructive version of the algebraic operation α.
- Computing the operation $\mathit{assume}[\varphi](a)$, which returns the most-precise abstraction of the set of stores represented by a for which a precondition φ holds. Thus, when applied to the most general abstract value $\top$, the procedure $\widehat{\mathit{assume}}[\varphi]$ computes $\widehat{\alpha}(\varphi)$. However, when applied to some other abstract value a, $\widehat{\mathit{assume}}[\varphi]$ refines a according to precondition φ. This is perhaps the most exciting application of the method described in the paper, because it would permit TVLA to be applied to large programs by using procedure specifications.
- Computing *best abstract transformers* for atomic program statements and conditions [2]. The current transformers in TVLA are conservative, but are not necessarily the best. Technically, the best abstract transformer of a statement described by a transformer τ amounts to $\mathit{assume}[\tau](a)$, where τ is a formula over the input and output states and a is the input abstract value. The method can also be used to compute best transformers for loop-free code fragments (i.e., blocks of atomic program statements and conditions).
- Computing the most-precise overapproximation of the meet of two abstract values. Such an operation is useful for combining forward and backward shape analysis to establish temporal properties, and when performing interprocedural analysis in the Sharir and Pnueli functional style [12]. Technically, the meet of abstract values a_1 and a_2 is computed by $\widehat{\alpha}(\widehat{\gamma}(a_1) \wedge \widehat{\gamma}(a_2))$.

The *assume* Operation can be used to perform interprocedural shape analysis using procedure specifications and assume-guarantee reasoning. Here the problem is to interpret a procedure's pre- and post-conditions in the most precise way (for a given abstraction). For every procedure invocation, we check if the current abstract value potentially violates the precondition; if it does, a warning is produced. At the point immediately after the call, we can assume that the post-condition holds. Similarly, when a procedure is analyzed, the pre-condition

[3] As a convention, a name of an operation marked with a "hat" ($\widehat{\ }$) denotes the algorithm that computes that operation.

is assumed to hold on entry, and at end of the procedure the post-condition is checked.

The core algorithm $\widehat{assume}$ presented in the paper computes $assume[\varphi](a)$, the refinement of an abstract value a according to precondition φ. In [16] we prove the correctness of the algorithm, i.e., $\widehat{assume}[\varphi](a) = assume[\varphi](a) = \alpha([\![\varphi]\!] \cap \gamma(a))$. Fig. 1 depicts the idea behind the algorithm. It shows the the concrete and abstract value-spaces as the rectangle on the left and the oval on the right. The points in the right oval represent abstract values with the corresponding sets of concrete values (defined by γ) shown as ovals on the left. The algorithm works its way down in the right oval, which on the left corresponds to progressing from the outer oval towards the inner region, labeled X. The algorithm repeatedly refines abstract value a by eliminating the ability to represent concrete stores that do not satisfy φ. It produces an abstract value that represents the tightest set of stores in $\gamma(a)$ that satisfy φ. Of course, because of the inherent loss of information due to abstraction, the result can also describe stores in which φ does not hold. However, the result is as precise as possible for the given abstraction, i.e., it is the tightest possible overapproximation to $[\![\varphi]\!] \cap \gamma(a)$ expressible in the abstract domain.

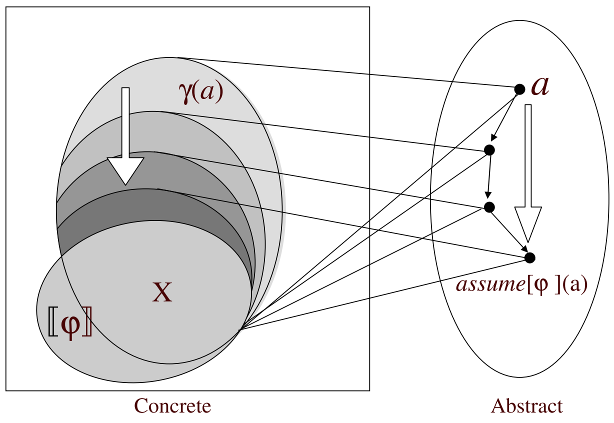


Fig. 1. The $\widehat{assume}[\varphi](a)$ algorithm. The set $X = [\![\varphi]\!] \cap \gamma(a)$ describes all stores that are represented by a and satisfy φ.

The $\widehat{assume}$ algorithm employs a decision procedure for the logic used to express properties of data structures. In [5], a logic named $\exists\forall^{DTC(E)}$ is described, which is both *decidable* and *useful* for reasoning about shape invariants. Its main features are sketched in Section 3.1. However, the $\widehat{assume}$ algorithm can also be used with an undecidable logic and a theorem prover; termination can be assured by using standard techniques (e.g., having the theorem prover return a safe answer if a time-out threshold is exceeded) at the cost of losing the ability to guarantee that a most-precise result is obtained.

Prototype Implementation To study the feasibility of our method, we have implemented a prototype of the $\widehat{assume}$ algorithm using the first-order theorem prover SPASS [14]. Because SPASS does not support transitive closure, the prototype implementation is applicable to shape-analysis algorithms that do not use transitive closure [6, 13]. So far, we tried three simple examples: two cases of $\widehat{assume}$, one of which is the running example of this paper, and one case of best transformer. On all queries posed by these examples, the theorem prover terminated. The number of calls to SPASS in the running example is 158, and the overall running time was approximately 27 seconds.

2 Overview of the Framework

This section provides an overview of the framework and the results reported in the paper. The formal description of the $\widehat{assume}$ algorithm appears in Section 3.

As an example, consider the following precondition, expressed in C notation as: `(x -> n == y) && (y != null)` (which will be abbreviated in this section as p), where `x` and `y` are program variables of the linked-list data-type defined in Fig. 2(a). The precondition p can be defined by a closed formula in first-order logic: $\varphi_0 \overset{\text{def}}{=} \exists v_1, v_2 : x(v_1) \wedge n(v_1, v_2) \wedge y(v_2)$. The operation $assume[p](a)$ enforces precondition p on an abstract value a. Typically, a represents a set of concrete stores that may arise at the program point in which p is evaluated. The abstract value a used in the running example is depicted by the graph in Fig. 2(S). This graph is an abstraction of all concrete stores that contain a non-empty linked list pointed to by `x`, as explained below.

2.1 3-Valued Structures

In this paper, abstract values that are used to represent concrete stores are sets of 3-valued logical structures over a vocabulary $\mathcal{P}$ of predicate symbols. Each structure has a universe U of individuals and a mapping ι from k-tuples of individuals in U to values $1, 0$, or $1/2$ for each k-ary predicate in $\mathcal{P}$. We say that the values 0 and 1 are **definite values** and that $1/2$ is an **indefinite value**, meaning "either 0 or 1 possible"; a value l_1 is **consistent** with l_2 (denoted by $l_1 \sqsubseteq l_2$) when $l_1 = l_2$ or $l_2 = 1/2$; $\bigsqcup W$ denotes the least upper bound of the values in the set W.

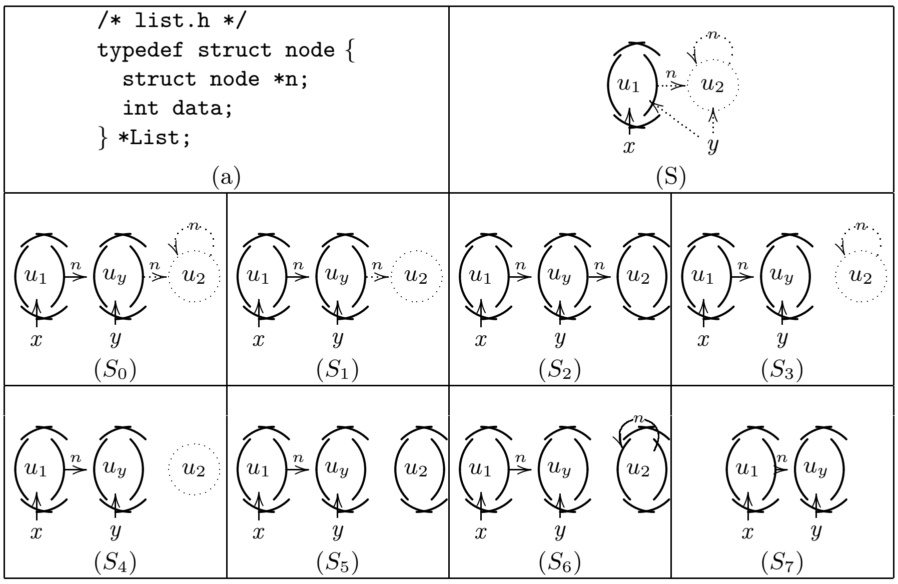
A 3-valued structure provides a representation of stores: individuals are abstractions of heap-allocated objects; unary predicates represent pointer variables that point from the stack into the heap; binary predicates represent pointer-valued fields of data structures; and additional predicates in $\mathcal{P}$ describe certain properties of the heap. A special predicate eq has the intended meaning of equality between locations. When the value of eq is $1/2$ on the pair $\langle u, u \rangle$ for some node u, then u is called a "summary" node and it may represent more than one linked-list element. Table 1 describes the predicates required for a program with pointer variables `x` and `y`, that manipulates the linked-list data-type defined in Fig. 2(a). 3-valued structures are depicted as directed graphs, with individuals as graph nodes. A predicate with value 1 is represented by a solid arrow; with value $1/2$ by a dotted arrow; and with value 0 by the absence of an arrow.

In Fig. 2(S), the solid arrow from x to the node u_1 indicates that predicate x has the value 1 for the individual u_1 in the 3-valued structure S. This means that any concrete store represented by S contains a linked-list element pointed to by program variable `x`. Moreover, it **must** contain additional elements (represented

Predicate	**Intended Meaning**
$x(v)$	Does pointer variable `x` point to element v?
$y(v)$	Does pointer variable `y` point to element v?
$n(v_1, v_2)$	Does the `n` field of v_1 point to v_2?
$eq(v_1, v_2)$	Do v_1 and v_2 denote the same element?
$is(v)$	Is v pointed to by more than one field ?

Table 1. The set of predicates for representing the stores manipulated by programs that use the `List` data-type from Fig. 2(a) and two pointer variables `x`, `y`.

by the summary node u_2, drawn as a dotted circle), some of which **may** be reachable from the head of the linked-list (as indicated by the dotted arrow from u_1 to u_2, which corresponds to the value 1/2 of predicate $n(u_1, u_2)$), and some of which **may** be linked to others (as indicated by the dotted self-arrow on u_2). The dotted arrows from y to u_1 and u_2 indicate that program variable `y` **may** point to any linked-list element. The absence of an arrow from u_2 to u_1 means that there is **no** n-pointer to the head of the list. Also, the unary predicate is is 0 on all nodes and thus not shown in the graph, indicating that every element of a concrete store represented by this structure may be pointed to by at most one n-field.



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

Fig. 2. (a) A declaration of a linked-list data-type in C. (S) The input abstract value $a = \{S\}$ represents all concrete stores that contain a non-empty linked list pointed to by the program variable `x`, where the program variable `y` may point to some element. (S_0–S_7) The result of computing $assume[p](a)$: the abstract value $a' = \{S_0, \ldots, S_7\}$ represents all concrete stores that contain a linked-list of length 2 or more that is pointed to by `x`, in which the second element is pointed to by `y`.

We next introduce the subclass of bounded structures [10]. Towards this end, we define **abstraction predicates** to be a designated subset of unary predicates, denoted by $\mathcal{A}$. In the running example, all unary predicates are defined as abstraction predicates. A **bounded structure** is a 3-valued structure in which for every pair of distinct nodes u_1, u_2, there exists an abstraction predicate q such that q evaluates to different definite values for u_1 and u_2. All 3-valued structures used throughout the paper are bounded structures. Bounded structures are used in shape analysis to guarantee that the analysis is carried out w.r.t. a finite set of abstract structures, and hence will always terminate.

2.2 Embedding Order on 3-Valued Structures

3-valued structures are ordered by the **embedding order** ($\sqsubseteq$), defined below. $S \sqsubseteq S'$ guarantees that the set of concrete stores represented by S is a subset of those represented by S'.

Let S and S' be two 3-valued structures, and let f be a surjective function that maps nodes of S onto nodes of S'. We say that f **embeds** S in S' (denoted by $S \sqsubseteq_f S'$) if for every predicate $q \in \mathcal{P}$ of arity k and all k-tuples $\langle u_1, \ldots, u_k \rangle$ in S, the value of q over $\langle u_1, \ldots, u_k \rangle$ is consistent with, but may be more specific than, the value of q over $\langle f(u_1), \ldots, f(u_k) \rangle$: $\iota^S(q)(u_1, \ldots, u_k) \sqsubseteq \iota^{S'}(q)(f(u_1), \ldots, f(u_k))$. We say that S **can be embedded into** S' (denoted by $S \sqsubseteq S'$) if there exists a function f such that $S \sqsubseteq_f S'$.

In fact, the requirement of $\mathit{assume}[p](a)$ can be rephrased using embedding: generate the most-precise abstract value a' such that all concrete stores that can be embedded into a' (i) can be embedded into a, and (ii) satisfy the precondition p. Indeed, the result of $\mathit{assume}[p](a)$, shown in Fig. 2(S_0–S_7), consists of 8 structures, each of which can be embedded into the input structure Fig. 2(S). The embedding function maps u_1 in the output structure to the same node u_1 in each of S_0–S_7 output structures. Each one of the output structures S_0–S_6 contains nodes u_y and u_2, both of which are mapped by the embedding to u_2 in S; for S_7, node u_y is mapped to u_2 in S. Thus, concrete elements represented by u_y and u_2 in the output structures are represented by a single summary node u_2 in the input structure. We say that node u_y is "materialized" from node u_2. As we shall see, this is the only new node required to guarantee the most-precise result, relative to the abstraction.

For each of $S_0, \ldots, S_7$, the embedding function described above is consistent with the values of the predicates. The value of x on u_1 is 1 in S_i and S structures. Indefinite values of predicates in S impose no restriction on the corresponding values in the output structures. For instance, the value of y is 1/2 on all nodes in S, which is consistent with its value 0 on nodes u_1 and u_2 and the value 1 on u_y in each of $S_0, \ldots, S_7$. The absence of an n-edge from u_2 back to u_1 in S implies that there must be no edge from u_y to u_1 and from u_2 to u_1 in the output structures, i.e., the values of the predicate n on these pairs must be 0.

2.3 Integrity Rules

A 2-valued structure is a special case of a 3-valued structure, in which predicate values are only 0 and 1. Because not all 2-valued structures represent valid stores, we use a designated set of **integrity rules**, to exclude impossible stores. The integrity rules are fixed for each particular analysis and defined by a conjunction of closed formulas over the vocabulary $\mathcal{P}$, that must be satisfied by all concrete structures. For the linked-list data-type in Fig. 2(a), the following conditions define the admissible stores: (i) each program variable can point to at most one heap node, (ii) the `n`-field of an element can point to at most one element, (iii) $is(v)$ holds if and only if there exist two distinct elements with n-fields pointing to v. Finally, eq is given the interpretation of equality: $eq(v_1, v_2)$ holds if and only if v_1 and v_2 denote the same element.

2.4 Canonical Abstraction

The abstraction we use throughout this paper is **canonical abstraction**, as defined in [11]. The surjective function β takes a 2-valued structure and returns a 3-valued structure with the following properties:

- β maps concrete nodes into abstract nodes according to **canonical names** of the nodes, constructed from the values of the abstraction predicates.
- β is a **tight** embedding [11], i.e., the value of the predicate q on an abstract node-tuple is 1/2 only when there exist two corresponding concrete node-tuples with different values.

A 3-valued structure S is an ICA (Image of Canonical Abstraction) if there exists a 2-valued structure $S^\natural$ such that $S = \beta(S^\natural)$. Note that every ICA is a bounded structure.

For example, all structures in Fig. 2(S_0–S_7) produced by $assume[p](a)$ operation are ICAs, whereas the structure in Fig. 2(S) is not an ICA. The structure in Fig. 2(S_1) is a canonical abstraction of the concrete structure in Fig. 3(a) and also the one in Fig. 3(b).

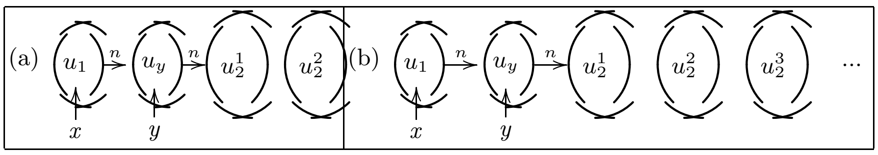


Fig. 3. Concrete stores represented by the structure S_1 from Fig. 2. (a) The concrete nodes u_2^1 and u_2^2 are mapped to the abstract node u_2. (b) The concrete nodes u_2^1, u_2^2 and u_2^3 are mapped to the abstract node u_2. More concrete structures can be generated in the same manner, by adding more isolated nodes that map to the summary node u_2.

The abstraction function α is defined by extending β pointwise, i.e., $\alpha(W) = \{\beta(S^\natural) \mid S^\natural \in W\}$ where W is a set of 2-valued structures. The concretization function γ takes a set of 3-valued structures W and returns a potentially infinite set of 2-valued structures $\gamma(W)$ where $S^\natural \in \gamma(W)$ iff $S^\natural$ satisfies the integrity rules and there exists $S \in W$ such that $\beta(S^\natural) \sqsubseteq S$.

The requirement of $assume[p](a)$ to produce the most-precise abstract value amounts to producing $\alpha(X)$, where X is the set of concrete structures that embed into a and satisfy p. Indeed, the result of $assume[p](a)$ in Fig. 2(S_0–S_7) satisfies this requirement, because S_0–S_7 are the canonical abstractions of all structures in X.

For example, structure S_1 from Fig. 2 is a canonical abstraction of each of the structures in Fig. 3. However, S_1 is not a canonical abstraction of S_2 from Fig. 2,[4] because the value 1/2 of n for $\langle u_y, u_2\rangle$ requires that a concrete structure abstracted by S_1 have two pairs of nodes with the same canonical names as $\langle u_y, u_2\rangle$ and with different values of n. This requirement does not hold in S_2, because it contains only one pair $\langle u_1, u_2\rangle$ with those canonical names. Without S_2, the result would not include the canonical abstractions of all concrete structures in X, but it would be semantically equivalent (because S_2 can be embedded into S_1). The version of the $\widehat{assume}[p](a)$ algorithm that we describe does include S_2 in the output. It is straightforward to generalize the algorithm to produce the smallest semantically equivalent set of structures.

It is non-trivial to produce the most-precise result for $assume[p](a)$. For instance, in each of S_0–S_6 there is no back-edge from u_2 to u_y even though both nodes embed into the node u_2 of the input structure, which has a self-loop with n evaluating to 1/2. It is a consequence of the integrity rules that no back-edge can exist from any u_2^j to u_y in any concrete structure that satisfies p: precondition p implies the existence of an n-pointer from u_1 to u_y, but u_y cannot have a second incoming n-edge (because the value of the predicate is on u_y is 0).

Consequently, to determine predicate values in the output structure, each concrete structure that it represents must be accounted for. Because the number of such concrete structures is potentially infinite, they cannot be examined explicitly. The algorithm described in this paper uses a decision procedure to perform this task symbolically.

Towards this end, the algorithm uses a symbolic representation of concrete stores as a logical formula, called a **characteristic formula**. The characteristic formula for an abstract value a is denoted by $\widehat{\gamma}(a)$; it is satisfied by a 2-valued structure $S^\natural$ if and only if $S^\natural \in \gamma(a)$. The $\widehat{\gamma}$ formula for shape analysis is defined in [15] for bounded structures, and it includes the integrity rules.

In addition, a necessary requirement for the output of $\widehat{assume}$ to be a set of ICAs is imposed by the formula $\varphi_{q,u_1,\ldots,u_k}$, defined in Eq. (1) below; this is used to check whether the value of a predicate q can be 1/2 on a node-tuple $\langle u_1, \ldots, u_k\rangle$ in a structure S. Intuitively, the formula is satisfiable when there exists a concrete structure represented by S that contains two tuples of nodes, both mapped to the abstract tuple $\langle u_1, \ldots, u_k\rangle$, such that q evaluates to different values on these tuples. If the formula is not satisfiable, S is not a result of canonical abstraction, because the value of q on $\langle u_1, \ldots, u_k\rangle$ is not as precise as possible, compared to the value of q on the corresponding concrete nodes.

3 The $\widehat{assume}$ Algorithm

The $\widehat{assume}$ algorithm is shown in Fig. 4. Section 3.1 explains the role of the decision procedure and the queries posed by our algorithm. The algorithm is explained in Section 3.2 (phase 1) and Section 3.3 (phase 2). Finally, the properties of the algorithm are discussed in Section 3.4.

[4] S_2 is a 2-valued structure, and is a canonical abstraction of itself.

```
procedure $\widehat{assume}(\varphi$: Formula, $a$: a set of bounded structures): Set of ICA structures
    $result := a$
    // Phase 1
    $result := \mathit{bif}\ (\varphi, result)$
    // Phase 2
    while there exists $S \in result, q \in \mathcal{P}$ of arity $k$, and $u_1, \ldots, u_k \in U^S$ such that
      $\iota^S(q)(u_1, \ldots, u_k) = 1/2$ and $done(S, q, u_1, \ldots, u_k) = false$ do
        $done(S, q, u_1, \ldots, u_k) := true$
        if $\widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u_1,\ldots,u_k}$ is not satisfiable then $result := result \setminus \{S\}$
        $S_0 := S[q(u_1, \ldots, u_k) \mapsto 0]$
        if $\widehat{\gamma}(S_0) \wedge \varphi$ is satisfiable then $result := result \cup \{S_0\}$
        $S_1 := S[q(u_1, \ldots, u_k) \mapsto 1]$
        if $\widehat{\gamma}(S_1) \wedge \varphi$ is satisfiable then $result := result \cup \{S_1\}$
    return $result$
```

Fig. 4. The $\widehat{assume}$ procedure takes a formula φ over the vocabulary $\mathcal{P}$ and computes the set of ICA structures *result*. $\widehat{\gamma}$ includes the integrity rules in order to eliminate infeasible concrete structures. The formula $\varphi_{q,u_1,\ldots,u_k}$ is defined in Eq. (1). The procedure *bif* (φ,result) is shown in Fig. 5. The flag $done(S, q, u_1, \ldots, u_k)$ marks processed q-tuples; initially, *done* is *false* for all predicate tuples.)

3.1 The Use of the Decision Procedure

The formula $\varphi_{q,u_1,\ldots,u_k}$ guarantees that a concrete structure must contain two tuples of nodes, both mapped to the abstract tuple $\langle u_1, \ldots, u_k\rangle$, on which q evaluates to different values. This is captured by the formula

$$\begin{aligned}\varphi_{q,u_1,\ldots,u_k} &\stackrel{\text{def}}{=} \exists w_1^1, \ldots, w_k^1, w_1^2, \ldots, w_k^2 : \textstyle\bigwedge_{i=1}^k \mathrm{node}_{u_i}^S(w_i^1) \wedge \bigwedge_{i=1}^k \mathrm{node}_{u_i}^S(w_i^2) \\ &\wedge \neg \textstyle\bigwedge_{i=1}^k eq(w_i^1, w_i^2) \wedge q(w_1^1, \ldots, w_k^1) \wedge \neg q(w_1^2, \ldots, w_k^2)\end{aligned} \tag{1}$$

$\varphi_{q,u_1,\ldots,u_k}$ uses the *node* formula, also defined in [15], which uniquely identifies the mapping of concrete nodes into abstract nodes. For a bounded structure S, $\mathrm{node}_u^S(v)$ simply asserts that u and v agree on all abstraction predicates.

The function **isSatisfiable(ψ)** invokes a decision procedure that returns `true` when ψ is satisfiable, i.e., the set of 2-valued structures that satisfy ψ is non-empty. This function guides the refinement of predicate values. In particular, the satisfiability checks on a formula ψ are used to make the following decisions:

- Discard a 3-valued structure S that does not represent any concrete store in X by taking $\psi \stackrel{\text{def}}{=} \widehat{\gamma}(S) \wedge \varphi$.
- Materialize a new node from node u w.r.t. the value of $q \in \mathcal{A}$ in S (phase 1) by taking $\psi \stackrel{\text{def}}{=} \widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u}$.
- Retain the indefinite value for predicate q on node-tuple $\langle u_1, \ldots, u_k\rangle$ in S (in phase 2) by taking $\psi \stackrel{\text{def}}{=} \widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u_1,\ldots,u_k}$.

This requires a decision procedure for the logic that expresses φ, $\varphi_{q,u}$ and $\widehat{\gamma}$, including the integrity rules.

A Decidable Logic for Shape Analysis [5] describes the logic $\exists\forall^{DTC(E)}$, defined by formulas of the form $\exists v_1, \ldots, v_n \forall v_{n+1}, \ldots, v_m : \varphi(v_1, \ldots, v_m)$, where

```
procedure bif ($\varphi$: Formula, $W$: Set of bounded structures): Set of bounded structures
for all $S \in W$
  if $\widehat{\gamma}(S) \wedge \varphi$ is **not** satisfiable then $W := W \setminus \{S\}$
while there exists $S \in W, q \in \mathcal{A}$ and $u \in U^S$ such that $\iota^S(q)(u) = 1/2$
  $W := W \setminus \{S\}$
  if $\widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u}$ is satisfiable then $W := W \cup S[u \mapsto u.0, u.1][q(u.0) \mapsto 0, q(u.1) \mapsto 1]$
  $S_0 := S[q(u) \mapsto 0]$
  if $\widehat{\gamma}(S_0) \wedge \varphi$ is satisfiable then $W := W \cup \{S_0\}$
  $S_1 := S[q(u) \mapsto 1]$
  if $\widehat{\gamma}(S_1) \wedge \varphi$ is satisfiable then $W := W \cup \{S_1\}$
return $W$
```

Fig. 5. The procedure takes a set of structures and a formula φ over the vocabulary $\mathcal{P}$, and computes the bifurcation of each structure in the input set, w.r.t. the input formula. Note that at the beginning of the procedure, it ensures that each structure in the working set W represents at least one concrete structure that satisfies φ. The formula $\varphi_{q,u}$ is defined in Eq. (1). The operation $S[u \mapsto u.0, u.1]$ performs a bifurcation of the node u in S, setting the values of all predicates on $u.0$ and $u.1$ to the values they had on u.

$\varphi(v_1, \ldots, v_m)$ is a quantifier-free formula over an arbitrary number of unary predicates and a single binary predicate $E(v_i, v_j)$. Instead of general transitive closure, $\exists\forall^{DTC(E)}$ only allows $E^*(v_i, v_j)$, which denotes the *deterministic transitive closure* [4] of E: E-paths that pass through an individual that has two or more successors are ignored in E^*. In [5], $\exists\forall^{DTC(E)}$ is shown to be useful for reasoning about shape invariants of data structures, such as singly and doubly linked lists, (shared) trees, and graph types [7]. Also, the satisfiability of $\exists\forall^{DTC(E)}$ formulas is decidable and NEXPTIME-complete, hence the $\exists\forall^{DTC(E)}$ decision procedure is a candidate implementation for the *isSatifiable* function. [5]

To sidestep the limitations of this logic, [5] introduces the notion of *structure simulation*, and shows that structure simulations can often be automatically maintained for the mutation operations that commonly occur in procedures that manipulate heap-allocated data structures. The simulation is defined via translation of FO^{TC} formulas to equivalent $\exists\forall^{DTC(E)}$ formulas.

Undecidable Logic The $\widehat{assume}$ algorithm can also be used with an undecidable logic and a theorem prover. The termination of the function isSatisfiable can be assured by using standard techniques (e.g., having the theorem prover return a safe answer if a time-out threshold is exceeded) at the cost of losing the ability to guarantee that a most-precise result is obtained.

If the timeout occurs in the first call to a theorem prover made by phase 2, the structure S is not removed from *result*. If a timeout occurs in any other satisfiability call made by *bif* or by phase 2, the structure examined by this call is added to the output set. Using this technique, $\widehat{assume}$ always terminates while producing sound results.

[5] Another candidate is the decision procedure for monadic 2-nd order logic over trees [3], MONA, which has non-elementary complexity.

3.2 Materialization

Phase 1 of the algorithm performs node “materialization” by invoking the procedure *bif*. The name *bif* comes from its main purpose: whenever a structure has an indefinite value of an abstraction predicate q on some abstract node, supported by different values on corresponding concrete nodes, the node is *bifurcated* into two nodes and q is set to different definite values on the new nodes. The *bif* procedure produces a set of 3-valued structures that have the same set of canonical names as the concrete stores that satisfy φ and embed into a. The *bif* procedure first filters out potentially unsatisfiable structures, and then iterates over all structures $S \in W$ that have an indefinite value for an abstraction predicate $q \in \mathcal{A}$ on some node u. It replaces S by other structures. As a result of this phase, all abstraction predicates have definite values for all nodes in each of the structures. Because the output structures are bounded structures, the number of different structures that can be produced is finite, which guarantees that *bif* procedure terminates.

In the body of the loop in *bif* , we check if there exists a concrete structure represented by S that satisfies φ in which q has different values on concrete nodes represented by u (the query is performed using the formula $\varphi_{q,u}$). In this case, a new structure S' is added to W, created from S by duplicating the node u in S into two instances and setting the value of q to 0 for one node instance, and to 1 for another instance. All other predicate values on the new node instances are the same as their values on u.

In addition, two copies of S are created with 0 and 1, respectively, for the value of $q(u)$. To guarantee that each copy represents a concrete structure in X an appropriate query is posed to the decision procedure. Omitting this query will produce a sound, but potentially overly-conservative result.

Fig. 6 shows a computation tree for the algorithm on the running example. A node in the tree is labeled by a 3-valued structure, sketched by showing its nodes. Its children are labeled by the result of refining the 3-valued structure w.r.t. the predicate and the node-tuple on the right, by the values shown on the outgoing edges.

The order in which predicate values are examined affects the complexity (in terms of the number of calls to a decision procedure, the size of the query formulas in each call and the maximal number of explored structures), but it does not affect the result, provided that all calls terminate. The order in Fig. 6 was chosen for convenience of presentation. The root of the tree contains the sketch of the input structure S from Fig. 2(S); u_1 is the left circle and u_2 is the right circle. Fig. 6 shows the steps performed by *bif* on the input $\{S\}$ in Fig. 2. *bif* examines the abstraction predicate y, which has indefinite values on the nodes u_1 and u_2. The algorithm attempts to replace S by T', T_1, and T_0, shown as the children of S in Fig. 6. The structures T' and T_1 are discarded because all of the concrete structures they represent violate integrity rule (i) for `x` (Section 2.3) and the precondition p, respectively. The remaining structure T_0 is further modified w.r.t. the value of $y(u_2)$. However, setting $y(u_2)$ to 0 results in a structure that does not satisfy p, and hence it is discarded.

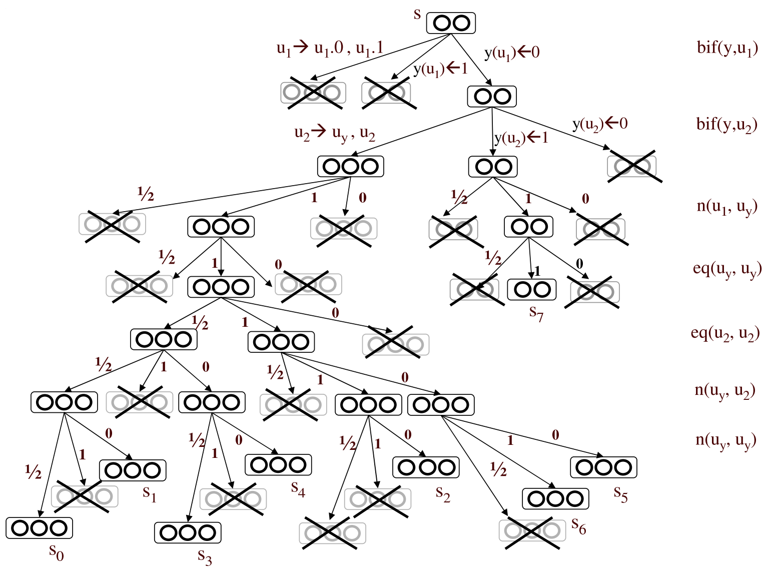


Fig. 6. A computation tree for $\widehat{assume}[p](a)$ for a shown in Fig. 2(a).

3.3 Refining Predicate Values

The second phase of the $\widehat{assume}$ algorithm refines the structures by lowering predicate values from 1/2 to 0 and 1, and throwing away structure S when the structure has a predicate q that has the value 1/2 for some tuple $q(u_1, \ldots, u_k)$, but the structure does not represent any 2-valued structure with corresponding tuples $q(u'_1, \ldots, u'_k) = 0$ and $q(u''_1, \ldots, u''_k) = 1$.

For each structure S and an indefinite value of a predicate $q \in \mathcal{P}$ on a tuple of abstract nodes, we eliminate structures in which the predicate has the same values on all corresponding tuples in all concrete structures that are represented by S and satisfy φ. (This query is performed using the formula in Eq. (1).) In addition, two copies of S are created with the values 0 and 1 for q, respectively. To guarantee that each copy represents a concrete structure in X, an appropriate query is posed to a decision procedure. The *done* flag is used to guarantee that each predicate tuple is processed only once.

The bulk of Fig. 6 (everything below the top two rows) shows the refinement of each predicate value in the running example. Phase 2 starts with two structures, T'_2 and T'_3, of size 2 and 3, produced by *bif*. Consider the refinement of T'_2 w.r.t. $n(u_1, u_y)$, where u_1 is pointed to by `x` and u_y is pointed to by `y` (the same node names as in Fig. 2).

The predicate tuple $n(u_1, u_y)$ cannot be set to 1/2, because it requires the existence of a concrete structure with two different pairs of nodes mapped to $\langle u_1, u_y\rangle$; however, integrity rule (i) in Section 2.3 implies that there is exactly one node represented by u_1 and exactly one node represented by u_y. Intuitively, this stems from the fact that the (one) concrete node represented by $u_1(u_y)$ is pointed to by `x(y)`. The predicate tuple $n(u_1, u_y)$ cannot be set to 0, because this violates the precondition p, according to which the element pointed to by `y` (represented by u_y) must also be pointed to by the n-field of the element pointed to by x (represented by u_1). Guided by the computation tree in Fig. 6, the reader can verify that the structures in Fig. 2(S_0–S_7) are generated by $\widehat{assume}[p](a)$. (The final answer is read out at the leaves).

3.4 Properties of the Algorithm

We determine the complexity of the algorithm in terms of (i) the size of each structure, i.e., the number of nodes and definite values, (ii) the number of structures, and (iii) the number of the calls to the decision procedure. The size of each query formula passed to the decision procedure is linear in the size of the examined structure, because $\widehat{\gamma}(S)$ is linear in S, φ is usually small, and the size of $\varphi_{q,u}$ is fixed for a given $\mathcal{P}$. The complexity in terms of (ii) and (iii) is linear in the height of the abstract domain of sets of ICA structures defined over $\mathcal{P}$, which is doubly-exponential in the size of $\mathcal{P}$. Nevertheless, it is exponentially more efficient than the naive **enumerate-and-eliminate** algorithm over the abstract domain. The reason is that the algorithm described in this paper examines only one descending chain in this abstract domain, as shown in Fig. 1.

To prove the correctness of the algorithm, it is sufficient to establish the following properties (the proofs appear in [16]):

1. All the structures explored by the algorithm are bounded structures.
2. $result \sqsupseteq \alpha([\![\varphi]\!] \cap \gamma(a))$. This requirement ensures that the result is **sound**, i.e., *result* contains canonical abstractions of all concrete structures in X. This is a global invariant throughout the algorithm.
3. $result \sqsubseteq \alpha([\![\varphi]\!] \cap \gamma(a))$. This requirement ensures that *result* does not contain abstract structures that are not ICAs of any concrete store in X. This holds upon the termination of the algorithm.

4 Computing the Best Transformer

The BT algorithm manipulates the two-store vocabulary $\mathcal{P} \cup \mathcal{P}'$, which includes two copies of each predicate — the original unprimed one, as well as a primed version of the predicate. The original version of the predicate contains the values before the transformer is applied, and the primed version contains the new values.

The best-transformer algorithm $BT(\tau, a)$ takes a set of bounded structures a over a vocabulary $\mathcal{P}$, and a transformer formula τ over the two-store vocabulary $\mathcal{P} \cup \mathcal{P}'$. It returns a set of ICA structures over the two-store vocabulary that is the canonical abstraction of all pairs of concrete structures $\langle S_1^\natural, S_2^\natural \rangle$ such that $S_2^\natural$ is the result of applying the transformer τ to $S_1^\natural$. $BT(\tau, a)$ is computed by $\widehat{assume}(\tau, extend(a))$ that operates over the two-store vocabulary, where $extend(a)$ extends each structure in $S \in a$ into one over a two-store vocabulary by setting the values of all primed predicates to 1/2.

The two-store vocabulary allows us to maintain the relationship between the values of the predicates before and after the transformer. Also, τ is an arbitrary formula over the two-store vocabulary; in particular, it may contain a precondition that involves unprimed versions of the predicates, together with primed predicates in the "update" part. The result of the transformer can be obtained from the primed version of the predicates in the output structure.

5 Related Work and Conclusions

In [9], we have presented a different technique to compute best transformers in a more general setting of finite-height, but possibly infinite-size lattices. The technique presented in [9] handles infinite domains by requiring that a decision procedure produce a concrete counter-example for invalid formulas, which is not required in the present paper.

Compared to [9], an advantage of the approach taken in the present paper is that it iterates from above: it always holds a legitimate value (although not the best). If the logic is undecidable, a timeout can be used to terminate the computation and return the current value. Because the technique described in [9] starts from $\bot$, an intermediate result cannot be used as a safe approximation of the desired answer. For this reason, the procedures discussed in [9] must be based on decision procedures. Another potential advantage of the approach in this paper is that the size of formulas in the algorithm reported here is linear in the size of structures (counting 0 and 1 values), and does not depend on the height of the domain.

This paper is also closely related to past work on predicate abstraction, which also uses decision procedures to implement most-precise versions of the basic abstract-interpretation operations. Predicate abstraction is a special case of canonical abstraction, when only nullary predicates are used. Interestingly, when applied to a vocabulary with only nullary predicates, the algorithm in Fig. 4 is similar to the algorithm used in SLAM [1]. It starts with 1/2 for all of the nullary predicates and then repeatedly refines instances of 1/2 into 0 and 1. The more general setting of canonical abstraction requires us to use the formula $\varphi_{q,u_1,u_2,\ldots,u_k}$ to identify the appropriate values of non-nullary predicates. Also, we need the first phase (procedure *bif*) to identify what node materializations need to be carried out.

This paper was inspired by the Focus[6] operation in TVLA, which is similar in spirit to the *assume* operation. The input of Focus is a set of 3-valued structures and a formula φ. Focus returns a semantically equivalent set of 3-valued structures in which φ evaluates to a definite value, according to the Kleene semantics for 3-valued logic [11]. The $\widehat{assume}$ algorithm reported in this paper has the following advantages: (i) it guarantees that the number of resultant structures is finite. The Focus algorithm in TVLA generates a runtime exception when this cannot be achieved. This make Focus a partial function, which was sometimes criticized by the TVLA user community. (ii) The number of structures generated by $\widehat{assume}$ is optimal in the sense that it never returns a 3-valued structure unless it is the canonical abstraction of some required store.

The latter property is achieved by using a decision procedure; in the prototype implementation, a theorem prover is used instead, which makes $\widehat{assume}$ currently slower than Focus. In the future, we plan to develop a specialized decision procedure for the logic $\exists\forall^{DTC(E)}$, which we hope will give us the benefits of $\widehat{assume}$ while maintaining the efficiency of Focus on those formulas for which Focus is defined.

[6] In Russian, Focus means "trick" like "Hocus Pocus".

To summarize, for shape-analysis problems, the methods described in this paper are more automatic and more precise than the ones used in TVLA, and allow modular analysis with assume-guarantee reasoning, although they are currently much slower. This work also provides a nice example of how abstract-interpretation techniques can exploit decision-procedures/theorem-provers. Methods to speed up these techniques are the subject of ongoing work.

References

1. T. Ball and S.K. Rajamani. The SLAM toolkit. In *Proc. Computer-Aided Verif.*, Lec. Notes in Comp. Sci., pages 260–264, 2001.
2. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.
3. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.
4. N. Immerman. *Descriptive Complexity.* Springer-Verlag, 1999.
5. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Decidable logics for expressing heap connectivity. In preparation, 2003.
6. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.
7. N. Klarlund and M. Schwartzbach. Graph types. In *Symp. on Princ. of Prog. Lang.*, New York, NY, January 1993. ACM Press.
8. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.
9. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Proc. VMCAI*, 2004. To appear.
10. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, pages 105–118, New York, NY, January 1999. ACM Press.
11. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 2002.
12. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
13. E. Y.-B. Wang. *Analysis of Recursive Types in an Imperative Language.* PhD thesis, Univ. of Calif., Berkeley, CA, 1994.
14. C. Weidenbach. SPASS: An automated theorem prover for first-order logic with equality. Available at "http://spass.mpi-sb.mpg.de/index.html".
15. G. Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2003. Available at "http://www.math.tau.ac.il/∼ gretay".
16. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. Technical report, TAU, 2003. Available at "http://www.cs.tau.ac.il/∼gretay".