Symbolically Computing Most-Precise Abstract Operations for Shape Analysis

G. Yorsh[*] T. Reps[†] M. Sagiv[*]

September 5, 2003

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 a shape descriptor (describing some set of concrete stores X) and a precondition p, and computes the most-precise shape descriptor for the stores in X that satisfy p. This algorithm solves several open problems in shape analysis: (i) computing the most-precise descriptor of a set of concrete stores represented 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 shape descriptors.

The algorithm employs 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.

The results indicate that the technique is feasible. We are currently developing a specialized decision procedure in order to conduct all the above operations on realistic programs.

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

[*]School of Comp. Sci., Tel-Aviv Univ., Tel-Aviv 69978, Israel. E-mail: {gretay, msagiv}@post.tau.ac.il.

[†]Comp. Sci. Dept., Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53706, USA. E-mail: reps@cs.wisc.edu.

certain correctness criteria [6]. The TVLA system [7] 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 $\gamma(a)$ 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 theorem provers. In a companion submission [16, 15], we show that the concretization of a shape descriptor can be expressed using a logical formula. Specifically, [16, 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 [1]. 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 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

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

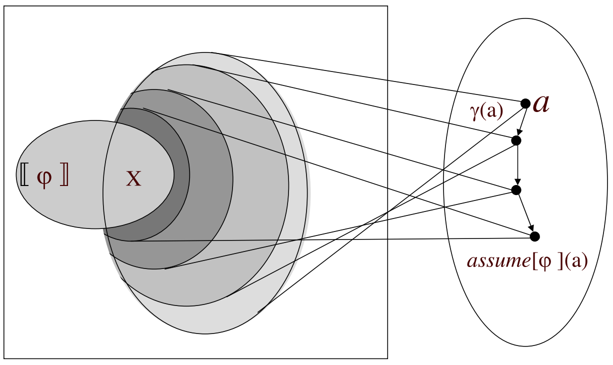


Figure 1: The $\widehat{assume}[\varphi](a)$ algorithm. The set $X = [\![\varphi]\!] \cap \gamma(a)$ describes all stores that are represented by a and satisfy φ.

loop-free code fragments (i.e., blocks of atomic program statements and conditions).

– Computing the most-precise overapproximation of the meet of two shape descriptors. 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 $\widehat{\alpha}(\widehat{\gamma}(a_1) \wedge \widehat{\gamma}(a_2))$.

1.1.1 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 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 the full paper we show that the algorithm is correct, i.e., $\widehat{assume}[\varphi](a) = assume[\varphi](a) = \alpha([\![\varphi]\!] \cap \gamma(a))$. Fig. 1 depicts the idea behind the algorithm. It repeatedly refines abstract value a by eliminating 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.

1.1.2 The Use of Decision Procedures

Our algorithms make calls to a theorem prover, to investigate the semantic properties of abstract values by checking for non-emptiness of sets of concrete stores described by formulas. However, in general, a theorem prover need not terminate, depending on the decidability of the query formulas passed to a theorem prover. The query formulas are conjuncts of (i) user-supplied specifications, such as pre- and post- conditions; (ii) a characteristic formula, that includes integrity rules; and (iii) the designated formulas that guide the refinement. The satisfiability queries for formula of type (i) not including the integrity rules, and for formulas of type (ii) are decidable, because these formulas are expressed using $\exists\forall$ subset of first-order logic. Thus, the termination depends on decidability of the logic to express the integrity rules and the specifications. There is a trade-off between the expressibility of the logic and its decidability. On one hand, the logic must be expressible enough to allow verification of non-trivial properties. On the other hand, the logic must be simple enough to allow decidability.

1.1.3 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 [5, 3, 13]. The termination of the algorithm depends on the termination of SPASS, which in turn depends on the decidability of the query formulas passed to a SPASS. The decidability issues, including transitive closure, are handled in a subsequent paper [4].

So far, we tried three simple examples: (i) assume of a precondition `(x->n == y) && (y != null)`, which is the running example of the paper, (ii) assume of a precondition `(x != null) && (y != null)` on the abstract value $\top$, and (iit) the best transformer for `y = x->n` on the abstract value $\top$. On all queries posed by these examples, the theorem prover terminated. The number of calls in the running example is 158, and the overall running time was approximately 27 seconds.

1.2 Outline of the Rest of the Paper

Section 2 presents the framework (its formal definition is in [11]) and the novel $\widehat{assume}$ algorithm at a semi-formal level. Section 3 presents the formal description of the $\widehat{assume}$ algorithm. Section 4 describes the three algorithms for computing basic operations used in shape analysis (best transformer, most-precise abstraction and meet), that employ $\widehat{assume}$ as a subroutine. Section 5 discusses related work. Section A provides the correctness proofs for $\widehat{assume}$.

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`.

2 Overview

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 precondition, syntactically written using C notations: `(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 $\varphi_0 \stackrel{\text{def}}{=} \exists v_1 v_2 : x(v_1) \wedge n(v_1, v_2) \wedge y(v_2)$ in first-order logic. The operation $assume[p](a)$ enforces a 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 when $l_1 = l_2$ or $l_2 = 1/2$.

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 a 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 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, that 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.

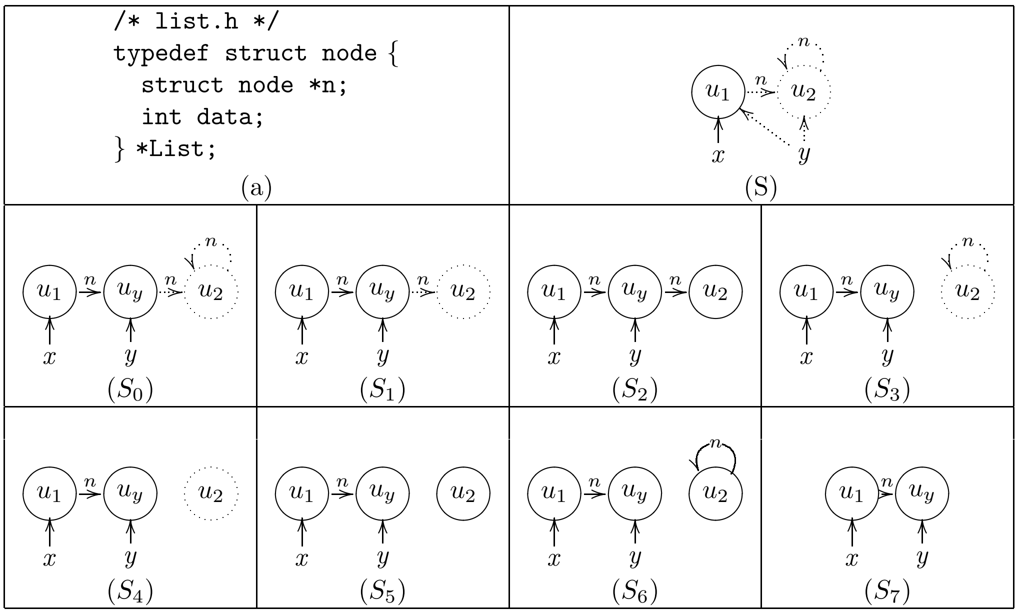
Any concrete store represented by the 3-valued structure in Fig. 2(S) includes a heap with at least one linked-list element (represented by u_1) that is pointed to by `x`. Moreover, the heap must contain additional elements (represented by u_2), some of which are linked to others, and some of which are reachable from the head of the linked-list.

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, which are the structures used in shape analysis to guarantee that the analysis is carried out w.r.t. a finite set of abstract structures, and hence would 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 $\langle f(u_1), \ldots, f(u_k) \rangle$. 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 $assume[p](a)$ can be rephrased using embedding: generate the most-precise abstract value a' that embeds all concrete stores that



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

Figure 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 $\mathit{assume}[p](a)$: 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`.

(i) can be embedded into a, and (ii) satisfy the precondition p.

Indeed, the result of $assume[p](a)$, 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 the input structure. 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. 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 will 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 $S_0, \ldots, S_6$. 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 [10]. The 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 $1/2$ when there exist two corresponding concrete node-tuples with different values. Intuitively, β is the embedding function that minimizes the information loss when multiple nodes of the 2-valued structure are mapped to the same abstract node.

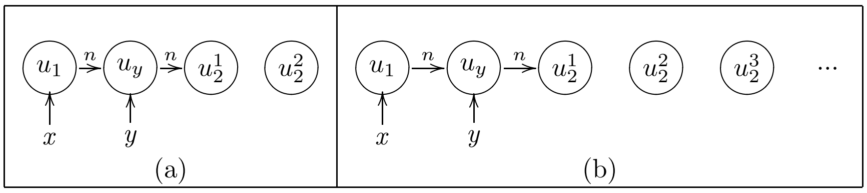


Figure 3: Concrete stores represented by the structure S_1 from Fig. 2. (a) The concrete nodes u_2^1 and u_2^2 are mapped to the abstract node u_2. (b) The concrete nodes u_2^1, u_2^2 and u_2^3 are mapped to the abstract node u_2. More concrete structures can be generated in the same manner, by adding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

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, and 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 in Fig. 3(b).

The β function of canonical abstraction serves as a representation function [8] to define our abstract domain. The abstract domain is a powerset of 3-valued structures, where the order relation on sets of structures is a Hoare order w.r.t. the embedding order on structures. The concrete domain is a powerset of all 2-valued structures that satisfy the integrity rules. Concrete and abstract domains are related by a Galois connection, in which the abstraction function α is defined by extending β pointwise, i.e., $\alpha(W) = \bigsqcup_{S^\natural \in W} \beta(S^\natural)$ 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 such that $S^\natural \in \gamma(W)$ iff $S^\natural$ satisfies the integrity rules and $\beta(S^\natural) \sqsubseteq W$:

$$\gamma(X) = \left\{ \begin{array}{ll} S \mid & S \text{ is 2-valued structure that satisfies the integrity rules} \\ & \hfill \text{and } \beta(S) \sqsubseteq X \end{array} \right\} \tag{1}$$

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,[2] 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

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

hold in S_2, because it contains only one pair $\langle u_1, u_2 \rangle$ with the same 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 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 theorem prover 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**. This provides a uniform way for expressing set of stores, program conditions and statements and procedure pre- and post-conditions, using logical 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, the necessary requirement for the output of $\widehat{assume}$ to be a set of ICAs is imposed by the formula $\varphi_{q,u_1,\ldots,u_k}$, explicitly defined in Eq. (2); it 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.

2.5 The *assume* Algorithm

The algorithm shown in Fig. 5 operates in two phases. The first phase causes "materialization" of nodes from summary nodes, until the manipulated structures have the same canonical names as the concrete stores that satisfy φ and embed into a. The second phase refines the structures by lowering predicate values from 1/2 to 0 and 1, and throws away structures with predicate value 1/2 for an (abstract) tuple that do not represent a concrete structure that has two corresponding tuples with different values of that predicate. at least one corresponding tuples with value 0 and at least one corresponding tuple with

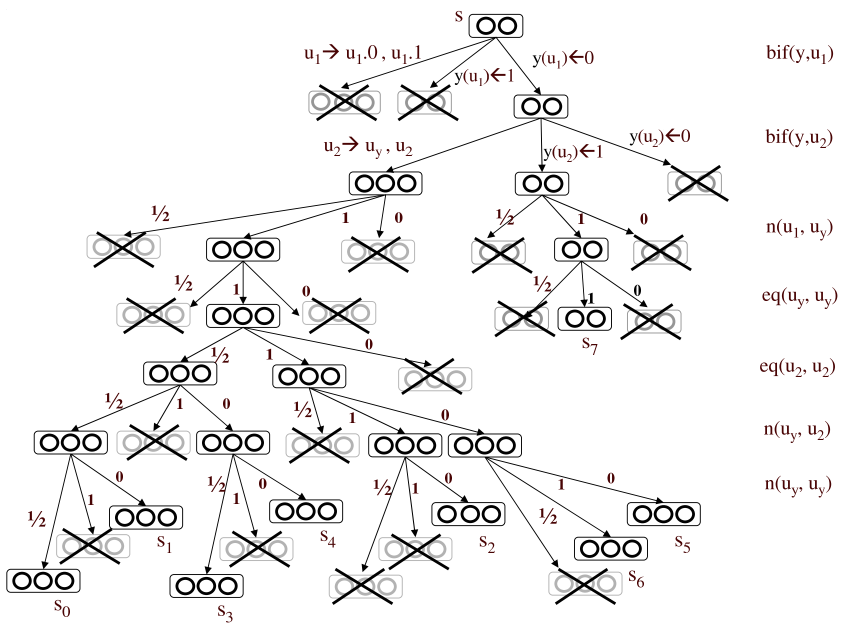


Figure 4: A computation tree for $\widehat{assume}[p](a)$ for a shown in Fig. 2(a).

value 1.

3 The *assume* Algorithm

The $\widehat{assume}$ algorithm is shown in Fig. 5. Section 3.1 explains the role of the theorem prover 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.

Fig. 4 shows a computation tree of the algorithm on our running example. A node in the tree is labelled by a 3-valued structure, sketched by showing its nodes. Its children are labelled 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 number of calls to a theorem prover and the maximal number of explored structures, but it does not affect the result. The order in Fig. 4 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.

3.1 The Use of the Theorem Prover

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

```
procedure $\widehat{assume}(\varphi$: Formula, $a$: a set of bounded structures): Set of ICA
    $result := a$
    // Phase 1
    $result := bif(\varphi, result)$
    // 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 $result$
```

Figure 5: 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. (2). The procedure *bif* (φ,result) is shown in Fig. 6. The flag $done(S, q, u_1, \ldots, u_k)$ marks processed values; initially, *done* is $false$ for all predicate tuples.)

evaluates to different values. This is captured by the formula

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

$\varphi_{q,u_1,\ldots,u_k}$ uses the *node* formula, also defined in [15], which uniquely identifies the mapping of concrete nodes into abstract nodes. For a bounded structure S and an abstract node u in S, $\text{node}_u^S(v)$ simply asserts that u and v agree on all abstraction predicates. Formally, the formula $\text{node}_u^S(v)$ is satisfied by a 2-valued structure $S^\natural$ and a concrete node $u^\natural$ in $S^\natural$ if and only if S is the canonical abstraction of $S^\natural$ ($\beta(S^\natural) = S$) and $u^\natural$ is mapped by canonical abstraction to u.

The function **isSatisfiable(ψ)** invokes a theorem prover which returns `true` when ψ is satisfiable, i.e., the set of 2-valued structures which satisfies ψ is non-empty. This function guides the refinement of predicate values. In particular, checking the satisfiability of ψ is used to make the following decisions:

– Discard a 3-valued structure S that does not represent any concrete store in X by taking $\psi \stackrel{\text{def}}{=} \varphi \wedge \widehat{\gamma}(S)$.

– Materialize a new node from the node u w.r.t. the value of $q \in \mathcal{A}$ in S (phase 1) by taking $\psi \stackrel{\text{def}}{=} \varphi \wedge \widehat{\gamma}(S) \wedge \varphi_{q,u}$.

– Retain indefinite value of a predicate q for node-tuple $\langle u_1, \ldots, u_k \rangle$ in S (in phase 2) by taking $\psi \stackrel{\text{def}}{=} \varphi \wedge \widehat{\gamma}(S) \wedge \varphi_{q,u_1,\ldots,u_k}$.

```
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 $\iota^S(q)(u) = 1/2$
        $W := W \setminus \{S\}$
        $S' := S[u \mapsto u.0, u.1][q(u.0) \mapsto 0, q(u.1) \mapsto 1]$ if $\widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u}$ is satisfiable then $W := W \cup \{S'\}$
        $S_0 := S[q(u) \mapsto 0]$
        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 $W$
```

Figure 6: 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 the procedure has a precondition, reinforced at the beginning, that each structure in the input set represents at least one concrete structure that satisfies φ. The formula $\varphi_{q,u}$ is defined in Eq. (2). The operation $S[u \mapsto u.0, u.1]$ performs a bifurcation of the node u in S, setting the values of all predicates except q on $u.0$ and $u.1$ to the values they had on u.

3.2 Materialization

Phase 1 of the algorithm performs node "materialization" by invoking the procedure *bif* . The name *bif* comes from it's main purpose: whenever a structure has an indefinite value of an abstraction predicate on some node, it *bifurcates* to a pair of structures for which the node has values 0 and 1, respectively, on that predicate. It 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 $S \in W$ with an indefinite value of 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. In addition, because the output structures are bounded structures, the number of different structures that can be produced is finite, which guarantees termination of *bif* procedure.

In the body of the loop, 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) value for $q(u)$. To guarantee that each copy represents a concrete structure in X an appropriate query is posed to a theorem prover. Omitting this query will produce a sound, but potentially overly-conservative result.

Fig. 4 shows the steps performed by *bif* on the input $\{S\}$ in Fig. 2. *bif* examines the abstraction predicate y, with indefinite values on the nodes u_1 and u_2. The algorithm attempts to replace S by S', S_1 and S_0, shown as the children of S in Fig. 4. The structures S' and S_1 are discarded because all concrete structures they represent violate the integrity rule (i) for `x` (Section 2.3) and the precondition φ_0, respectively. The remaining structure S_0 is further modified w.r.t. $y(u_2)$. However, setting $y(u_2)$ to 0 results in a structure that does not satisfy φ_0, and hence is discarded.

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 with value 1/2 of some predicate for a tuple that do not represent any 2-valued structure that has two corresponding tuples with different value of that predicate.

For each structure and an indefinite value of a predicate $q \in \mathcal{P}$ on an abstract node tuple, 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 (the query is performed using the formula in Eq. (2)). In addition, two copies of S are created with 0,(and 1) value for q. To guarantee that each copy represents a concrete structure in X, an appropriate query is posed to a theorem prover. The *done* flag is used to guarantee that each predicate tuple is processed only once.

Fig. 4 shows the refinement of each predicate value in the running example. Phase 2 starts with two structures, S_2' and S_3', of size 2 and 3, produced by *bif* . Consider the refinement of S_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` (same node names as in Fig. 2).

The predicate $n(u_1, u_y)$ cannot be set to 1/2, because it requires the existence of a concrete structure with two different pairs of nodes mapped to $\langle u_1, u_y \rangle$, but the 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 concrete nodes represented by $u_1(u_y)$ is pointed to by `x`(`y`). The predicate $n(u_1, u_y)$ cannot be set to 0, because it violates the precondition φ_0, according to which the element pointed to by `y` (represented by u_y) must also be pointed to by n-field from the element pointed to by `x` (represented by u_1). Guided by the computation tree in Fig. 4, the reader can verify that the structures in Fig. 2(S_0–S_7) are generated by $\widehat{assume}[p](a)$ (read out the leaves).

3.4 Properties of the Algorithm

3.4.1 Timeout of 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 call made by *bif* or by phase 2, the structure examined by this call is added to the output set. Using this technique the methods always terminate while producing sound results.

3.4.2 Termination

Assuming that all calls to the theorem prover terminate, the algorithm always terminates. The first phase (*bif*) terminates, because in each iteration the number of indefinite values per structure is reduced. The reason is that in each iteration a structure can be replaced by at most 3 structures, each of which contains a smaller number of indefinite values. The *bif* algorithm terminates when the working set W does not contain abstraction predicates with indefinite values.

Similar arguments show that the second phase terminates, as follows. If the algorithm retains an indefinite value of a predicate, this value is marked by *done* flag and not changed in subsequent iterations. In each iteration, a structure in $result$ can be replaced by at most 3 structures, each of which contains smaller number of indefinite values, that are not marked with *done*. The algorithm terminates when no unmarked indefinite values remain in $result$.

3.4.3 Complexity

We determine the complexity of the algorithm in terms of (i) the number of nodes in each structure, (ii) the number of structures, and (iii) the number of the calls to a theorem prover. The complexity in terms of (ii) and (iii) is linear in the height of the abstract domain of all sets of ICA 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 be more precise, at the end of *bif*, all abstraction predicates are definite, which allows at most $2^{|A|}$ different canonical names. Thus, number of structures with different canonical names that can possible be generated by *bif* is bounded by $2^{2^{|A|}}$. For each of these structures, each predicate has 3 possible values for each node-tuple. As mentioned earlier, the order in which predicate values are examined effects the number of calls to a theorem prover. In the worst case, number of calls in *bif* is bounded by $2^{3^{|A|}}$ and in phase 2 by $(2^{P_0} + 3^{|P_1| \times 2^{|A|} + |P_2| \times (2^{|A|})^2}) \times 2^{2^{|A|}}$ where $|A|$ is the number of abstraction predicates in $\mathcal{P}$, and P_i is the number of non-abstraction predicates of size i in $\mathcal{P}$, assuming that the maximal arity of a predicate in $\mathcal{P}$ is 2.

3.4.4 Correctness

The algorithm $\widehat{assume}[\varphi](a)$ computes the operation $assume[\varphi](a)$, defined by $\alpha(X)$, where $X \overset{\text{def}}{=} [\![\varphi]\!] \cap \gamma(a)$. To show the correctness of the $\widehat{assume}$ algorithm, i.e. $result = \alpha(X)$, it is sufficient to establish the following properties:

1. $result \sqsupseteq \alpha(X)$. This requirement ensures that the result is **sound**, i.e., $result$ contains canonical abstractions of all concrete structures in X. In other words, for all $S^\natural \in X$ there exists $S \in result$ such that $\beta(S^\natural) \sqsubseteq S$. Because S is a bounded structure it is sufficient to show that $S^\natural \sqsubseteq S$. This is a global invariant throughout the algorithm, proved using induction, as shown in Lemma A.1

2. $result \sqsubseteq \alpha(X)$. This requirement ensures that all 3-valued structures in $result$ are ICAs of a concrete structure in X. This holds only upon the termination of the algorithm. From Lemma A.3 it follows that for each $S \in result$ there exists a concrete structure $S^\natural \in X$ that is embedded into S. Lemma A.5 shows that the embedding is tight, i.e., $\beta(S^\natural) = S$.

Because the algorithm uses $\widehat{\gamma}$ operation, defined only for bounded structure, Lemma A.6 shows that all the structures explored by the algorithm are bounded structures.

4 Applications

This section shows how $\widehat{assume}$ can be used to implement algorithms for computing the best transformer (Section 4.1), the abstraction of potentially infinite set of stores α (Section 4.2) and the meet of two abstract values (Section 4.3).

4.1 Computing the Best Transformer

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 two-store vocabulary by setting the values of all primed predicates to 1/2.

The BT algorithm manipulates the two-store vocabulary $\mathcal{P} \cup \mathcal{P}'$, which includes two copies of each predicate — the original, unprimed and the 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. This allows us to maintain the relationship between the values of the predicates before and after the transformer. Note that the set $\mathcal{A}$ contains both unprimed and primed versions of the abstraction predicates, and the integrity rules contain multiple copies of each rule — for primed and unprimed versions of all predicates used in a rule. Also, τ is an arbitrary formula over the two-store vocabulary; in particular, it may contain a precondition, that involves unprimed version 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, while the procedure refines both primed and unprimed predicates. If the values of the unprimed (original) predicates are not important to the subsequent analysis, primed non-abstraction predicates need not be refined. This sound optimization may be significant when unprimed predicate values become definite as a result of a precondition, for example.

4.2 Computing α for Shape Analysis

Given a specification of concrete stores as a logical formula φ, we can compute $\alpha([\![\varphi]\!])$, denoted by $\widehat{\alpha}(\varphi)$ by simply calling $\widehat{assume}[\varphi](\top)$.

The abstract value $\top$ describes all concrete stores, i.e., for all 2-valued structure $S^\natural$ that satisfies the integrity rules, $S^\natural \in \gamma(\top)$. Therefore, the result of $\widehat{assume}[\varphi](\alpha)$ defined by $\alpha([\![\varphi]\!] \cap \gamma(\top))$ is exactly $\alpha([\![\varphi]\!])$. Operationally, $\top = \{S_{empty}, S_{summary}\}$, where S_{empty} is a structure with an empty universe and all nullary predicates set to 1/2, and $S_{summary}$ is a structure with exactly one summary node, and all predicate values are 1/2.

4.3 Computing Meet for Shape Analysis

A meet operation takes two abstract values and returns the canonical abstraction of concrete stores represented by the both inputs. It can be computed using $\widehat{\gamma}$ and $\widehat{\alpha}$. Given two sets of bounded structures a_1 and a_2 over the vocabulary $\mathcal{P}$, generate the characteristic formula for each of the sets and use $\widehat{\alpha}$ procedure as a subroutine. The definition of $\widehat{\gamma}$ guarantees that the result of $\widehat{\alpha}(\widehat{\gamma}(a_1) \wedge \widehat{\gamma}(a_2))$ is $\alpha(\gamma(a_1) \cap \gamma(a_2))$ as required.

5 Related Work and Conclusions

In a companion submission, we present a different technique to compute best transformers [9] 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 theorem prover produces a concrete counter-example for invalid formulas, which is not required in the present paper. The present paper presents techniques that apply to canonical-abstraction domains. In contrast, [9] is not restricted to canonical-abstraction domains; for instance, it applies to some abstract domains of infinite size (although they still must be of finite height).

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 $\perp$, 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. 5 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 materialization needs to be carried out.

This paper was inspired by the Focus[3] operation in TVLA, which 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 an 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 a canonical abstraction of some required store. The latter property is achieved by using a theorem prover, which in the current implementation makes $\widehat{assume}$ slower than Focus. In the future, we plan to develop specialized theorem provers that will give us the benefits of $\widehat{assume}$ while maintaining the efficiency of Focus on those formulas for which Focus is defined.

To summarize, for problems that can be addressed via first-order shape analysis, the methods described in this paper are more automatic and more precise than the ones used in TVLA, although they are currently much slower. This provides a nice example of how abstract-interpretation techniques can exploit theorem provers. Methods to speed up these techniques are the subject of ongoing work.

References

[1] SLAM toolkit. Available at "http://research.microsoft.com/slam/".

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

[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] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 28–40, New York, NY, 1989. ACM Press.

[4] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Decidable logics for expressing heap connectivity. In preparation, 2003.

[5] 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.

[6] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Proc. of the Int. Symp. on Software Testing and Analysis*, pages 26–38, 2000.

[7] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.

[8] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[9] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. Tech. Rep. TR-1468, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, January 2003. Available at "http://www.cs.wisc.edu/wpis/papers/tr1468.ps".

[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, M. Sagiv, and R. Wilhelm. Logical characterization of heap abstractions. Submitted for publication, 2003.

A Correctness Proofs

Lemma A.1 *At each step of $\widehat{assume}$, the following holds. If $S^\natural \in X$ then there exists $S \in result$ such that $S^\natural \sqsubseteq S$.*
Proof: Use induction on the value of *result* at each step of $\widehat{assume}$.

The base case: after the initialization phase $result_0 = a$, therefore it represents all concrete stores in $\gamma(a)$, in particular all those that satisfy φ.

The induction step: Let $S^\natural$ be a concrete structure such that $S^\natural \in X$. Assume that after i steps of the procedure, the hypothesis holds: there exists $S_i \in result_i$ such that $S^\natural \sqsubseteq S_i$. Show that after step $i+1$, there exists $S_{i+1} \in result_{i+1}$ such that $S^\natural \sqsubseteq S_{i+1}$.

If step $i+1$ is the call to the procedure $bif(result)$, the conclusion is obtained from Lemma A.2, because all concrete structures satisfying φ that are represented by an abstract structure, are also represented by a bifurcation of the abstract structure — there is no loss of "important" structures during bifurcation.

Otherwise, step $i+1$ is an operation performed during the inner loop of phase 2. Suppose that it operates with a structure S, predicate q of arity k and node tuple $\langle u_1, \ldots, u_k \rangle$ in S. The only structure that could be removed from *result* in this step is S.

Recall that S_i denotes the structure in $result_i$ that, by assumption, represents $S^\natural$. If the structure S, that can be removed from *result*, is **not** S_i, the hypothesis holds for $i+1$ and S_i is the structure that represents $S^\natural$ in $result_{i+1}$, i.e., S_{i+1} is S_i. Otherwise, S and S_i is the same structure, thus there exists an embedding function g such that $S^\natural \sqsubseteq_g S$. We shall prove that if S is removed from *result*, then one of the structures S_0 or S_1 represents $S^\natural$ and it is added to *result*.

According to the algorithm, S is removed when all concrete structures represented by S that satisfy φ have the same value for all node tuples mapped to $\langle u_1, \ldots, u_k \rangle$ by the embedding. In particular, this holds for $S^\natural$. Without loss of generality, assume that the value is 0 and show that $S^\natural$ is embedded in S_0. The embedding function f such that $S^\natural \sqsubseteq_f S_0$ is g: (i) because S and S_0 have the same universe, f is well-defined and surjective; (ii) we only have to show that f preserves values of q over $\langle u_1, \ldots, u_k \rangle$, because the values of other predicates are the same in S and S_0. The value of q over all tuples mapped to $\langle u_1, \ldots, u_k \rangle$ is 0 by assumption, and $\iota^{S_0}(q)(u_1, \ldots, u_k)$ is 0 by the construction of S_0.

To complete the proof, we have to show that S_0 is added to *result*, that is the if-condition that guards the statement $result := result \cup \{S_0\}$ is true. We have to show that there exists a concrete structure that satisfies the formula $\widehat{\gamma}(S_0) \wedge \varphi$. Indeed, $S^\natural$ satisfies the condition: $S^\natural$ satisfies $\widehat{\gamma}(S_0)$ because it is embedded into S_0 as shown above; also, by assumption, $S^\natural$ satisfies φ.

Lemma A.2 *Consider an iteration of the while-loop in bif procedure. Let $S \in W$, q be an abstract predicate and $u \in U^S$ handled in that iteration. Let $S^\natural$ be a concrete structure such that $S^\natural \in X$ and $S^\natural$ is embedded into S. $S^\natural$ is embedded into one of the structures $\{S', S_0, S_1\}$, denote it by S''.*
Proof: By assumption, there exists embedding function f such that $S^\natural \sqsubseteq_f S$. Show that there exists $S'' \in \{S', S_0, S_1\}$ such that $S^\natural$ is embedded into S'' by constructing an embedding function $f' \colon S^\natural \mapsto S''$, based on f.

– If $S^\natural \models \widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u}$ then $S^\natural$ contains two nodes, denoted by u_0 and u_1, such that the value of q on u_0 is 0 and the value of q on u_1 is 1. In this case, $S^\natural$ is embedded into S' using the following embedding function:

$$f'(u^\natural) = \begin{cases} u.0 & \text{if } f(u^\natural) = u \text{ and } \iota^{S^\natural}(q)(u) = 0 \\ u.1 & \text{if } f(u^\natural) = u \text{ and } \iota^{S^\natural}(q)(u) = 1 \\ f(u^\natural) & \textit{otherwise} \end{cases}$$

  f' is well-formed because f is and $\iota^{S^\natural}(q)(u)$ cannot be 0 and 1 simultaneously. f' is surjective: its image includes $u.0$ and $u.1$, because $f'(u_0) = u.0$ and $f'(u_1) = u.1$ as follows for the denotations above; other elements of S' are images of f', because f' is the same as f and f is surjective, by assumption.

  Show that f' preserves predicate values. The values of all predicates on all tuples in S', are the same as in S, except the value of q on the new nodes $u.0$ and $u.1$. f' preserves these values, because f does and f' is the same as f for the relevant nodes (these are the concrete nodes in $S^\natural$ that are **not** mapped to the new nodes of S'). Let $u^\natural \in S^\natural$ such that $f(u^\natural) = u$. Without loss of generality, let $\iota^{S^\natural}(q)(u^\natural) = 0$. By definition of f', $f'(u^\natural) = u.0$. By definition of S', the value of q on $u.0$ is 0, that is the same as the value of q on $u^\natural$. It shows that f' preserves the values of q. The case where $\iota^{S^\natural}(q)(u^\natural) = 1$ is symmetric.

– If $S^\natural$ does not satisfy $\widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u}$ then the value of q on all nodes in $S^\natural$ that are mapped to u by the embedding is the same. If this value is 0, S is embedded into S_0, otherwise — into S_1. These cases are symmetric, therefore we consider only the former. Note that S and S_0 have the same universe, and differ only in the value of q on u. Hence, the embedding function $f' \colon S^\natural \mapsto S_0$ is the same as f. f' is well-formed and surjective because f is. For all predicate values, except the value of q on u, f' preserves the values of the predicates, because these values are the same in S and S'. The value of q on u in S_0 is 0, by construction of S_0. The value of q on all nodes in $S^\natural$ that are mapped to u by f' is 0, by assumption. Therefore, the value of q is preserved by f'.

Lemma A.3 *At each step of $\widehat{assume}$, the following holds. For each structure $S \in result$ there is a concrete structure $S^\natural \in X$ that embeds into S.*
Proof: By induction on the steps of $\widehat{assume}$ procedure.

The base case: After checking the precondition, each structure retained in W represents at least one concrete store that satisfies φ, because $\widehat{\gamma}(S) \wedge \varphi$ is satisfiable. Thus, the claim holds for all structures in W.

The induction step:
First, show that the claim holds on each iteration of *bif* procedure. Consider an iteration that operates on some structure $S \in W$. By inductive assumption that holds at the beginning of this iteration, there exists a concrete structure $S^\natural \in X$ such that $S^\natural \sqsubseteq S$. There are three cases in which a structure might be added to W.

Suppose that S' is added to W. In this case, there exists a concrete structure that satisfies $\widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u}$, denote it by $S^\natural$. Consequently, $S^\natural$ satisfies φ and embeds into S. Using Lemma A.2, $S^\natural$ is embedded into S', proving the claim.

Suppose that S_0 is added to W in statement $W := W \cup \{S_0\}$. This statement is executed when the if-condition that guards it is true, i.e, there exists a concrete structure that satisfies $\widehat{\gamma}(S_0) \wedge \varphi$. In particular, this concrete structure satisfies φ and embeds into S_0, and thus embeds into the input a of $\widehat{assume}$, as follows from Lemma A.4, proving the claim. The third case, in which S_1 is added to X, is symmetric to this case.

Show that in phase 2 of $\widehat{assume}$, the claim holds in each iteration of the loop. A structure S_0 (S_1) can be inserted into *result* only when the if-condition that guards the insertion statement is true. The if-condition requires that there exists a concrete structure $S^\natural$ that satisfies φ and embeds into S_0 (S_1). According to the algorithm, $S_0 \sqsubseteq S$ ($S_1 \sqsubseteq S$). By Lemma A.4, $\{S\} \sqsubseteq a$. To summarize, $\{S^\natural\} \sqsubseteq \{S_0\} \sqsubseteq \{a\}$ and $S^\natural \models \varphi$, hence $S^\natural \in X$, which proves this lemma.

Lemma A.4 *At each step of $\widehat{assume}$, the following holds. If $S \in result$ then there exists $S' \in a$ such that $S \sqsubseteq S'$.*
Proof: By induction on the steps of the algorithm.

The base case: After initialization, *result* equals to a.

The induction step: Show that after each step of the algorithm, the hypothesis holds.

In phase 1, let S be the structure removed from W in some iteration of *bif* , that operates over a predicate q and node u in S. By the inductive assumption, $\{S\} \sqsubseteq a$. If S_0 (S_1) is added to W, then the hypothesis holds, because $S_0 \sqsubseteq S$ ($S_1 \sqsubseteq S$) trivially using and identity function. If S' is added to W, hypothesis also holds, because $S' \sqsubseteq S$ using the embedding function f':

$$f'(u') = \begin{cases} u & \text{if } u' = u.0 \text{ or } u' = u.1 \\ u' & \textit{otherwise} \end{cases}$$

In phase 2 of $\widehat{assume}$, let S be a structure examined in some iteration of the loop. By the inductive assumption, $\{S\} \sqsubseteq a$. The only structures that might be added to *result* are S_0 and S_1. According to the algorithm, S_0 (S_1) differs from S by a one predicate value, that is 0 (1) in S_0 and 1/2 in S. Therefore $S_0 \sqsubseteq S$ ($S_1 \sqsubseteq S$) using the identity embedding function, which completes the proof.

Lemma A.5 *Consider the content of the set result at the end of $\widehat{assume}$. If $S \in result$ then there exists $S^\natural$ such that $S^\natural \in X$ and $\beta(S^\natural) = S$.*
Proof: For the sake of argument, assume that there exists $S \in result$ such that for all concrete structures $S^\natural \in X$ that embed into S, $\beta(S^\natural) \neq S$.

Recall that at the end of *bif* procedure, all abstraction predicates have definite values in S. During phase 2, predicate values can only be lowered, meaning that the abstraction predicates remain definite. According to Lemma A.3, there exists a concrete structure $S^\natural$ such that $S^\natural \sqsubseteq S$. Because embedding preserves canonical names and all abstract predicates in S are definite, the embedding is possible only if the set of canonical names in $S^\natural$ and S is the same. Consequently, $\beta(S^\natural)$ embeds into S using an **identity** function. Therefore, the assumption $\beta(S^\natural) \neq S$ implies that there is a predicate with an indefinite value in S, but with a definite value in $\beta(S^\natural)$.

Formally, for each concrete structure in X that embeds into S, there exists a predicate q with an indefinite value on some tuple of nodes $\langle u_1, \ldots, u_k\rangle$ in S, such that the value of q on all tuples of nodes in the concrete structure $S^\natural$ that are mapped to $\langle u_1, \ldots, u_k\rangle$ by the embedding, is the same. This assumption means that in phase 2 of $\widehat{assume}$, when the value of q on $\langle u_1, \ldots, u_k\rangle$ in S is examined, the first if-condition is true, because the formula $\widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u_1,\ldots,u_k}$. Therefore, the statement guarded by this if-condition is executed, removing the structure S from *result* set. Therefore, a contradiction is obtained.

Lemma A.6 *Every step of $\widehat{assume}$ algorithm, the result is a bounded structure, given that the input is a bounded structure. This property is a precondition for the use of $\widehat{\gamma}$.*
Proof: Assume that the input of each operation considered below is a bounded structure. Then, to violate this "boundedness" property, the operation must change a definite value of some abstraction predicate, according to the definition of a bounded structure. (from 1 to 0 or 1/2 and from 0 to1 or 1/2).

The procedure *bif* either (i) lowers a value of a predicate from 1/2 to 1 or 0, or (ii) duplicates a node and sets an abstraction predicate q with indefinite value to definite values on the two copies of the node. Both operations do not violate "boundedness" property. Also, phase 2 of the algorithm by its definition can only lower predicate values, therefore it cannot violate the "boundedness" property.