# A Relational Approach to Interprocedural Shape Analysis

BERTRAND JEANNET
INRIA
ALEXEY LOGINOV
GrammaTech, Inc.
THOMAS REPS
University of Wisconsin
and
MOOLY SAGIV
Tel Aviv University

This article addresses the verification of properties of imperative programs with recursive procedure calls, heap-allocated storage, and destructive updating of pointer-valued fields, that is, *interprocedural shape analysis*. The article makes three contributions.

—It introduces a new method for abstracting relations over memory configurations for use in abstract interpretation.

—It shows how this method furnishes the elements needed for a compositional approach to shape analysis. In particular, abstracted relations are used to represent the shape transformation performed by a sequence of operations, and an overapproximation to relational composition can be performed using the meet operation of the domain of abstracted relations.

—It applies these ideas in a new algorithm for context-sensitive interprocedural shape analysis. The algorithm creates procedure summaries using abstracted relations over memory configurations, and the meet-based composition operation provides a way to apply the summary transformer for a procedure $P$ at each call site from which $P$ is called.

A preliminary version of this article appeared in the *Proceedings of the 11th International Static Analysis Symposium* (*SAS'04*) 2004 [Jeannet et al. 2004].
This work was supported in part by ONR under grants N00014-01-1-0796 and N00014-01-1-0708, and by NSF under grants CCR-9986308, CCF-0540955, and CCF-0524051.
When the research in this article was carried out, B. Jeannet was affiliated with INRIA or visiting the University of Wisconsin, and A. Loginov was affiliated with the University of Wisconsin.
Authors' addresses: B. Jeannet, INRIA; email: Bertrand.Jeannet@inrialpes.fr; A. Loginov, GrammaTech, Inc.; email: alexey@grammatech.com; T. Reps, Computer Science Department, University of Wisconsin and GrammaTech, Inc.; email: reps@cs.wisc.edu; M. Sagiv; School of Computer Science, Tel Aviv University; email: msagiv@post.tau.ac.il.

DOI 10.1145/1667048.1667050 http://doi.acm.org/10.1145/1667048.1667050

The algorithm has been applied successfully to establish properties of both (i) recursive programs that manipulate lists and (ii) recursive programs that manipulate binary trees.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures; dynamic storage management, procedures, functions and subroutines; recursion*; E.1 [**Data**]: Data Structures—*Lists, stacks and queues, trees*; E.2 [**Data**]: Data Storage Representations—*Composite structures; linked representations*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions; invariants*

General Terms: Algorithms, Languages, Theory, Verification

Additional Key Words and Phrases: Abstract interpretation, context-sensitive analysis, interprocedural dataflow analysis, destructive updating, pointer analysis, shape analysis, static analysis, 3-valued logic

**ACM Reference Format:**

Jeannet, B., Loginov, A., Reps, T., and Sagiv, M. 2010. A relational approach to interprocedural shape analysis. ACM Trans. Program. Lang. Syst. 32, 2, Article 5 (January 2010), 52 pages.
DOI = 10.1145/1667048.1667050 http://doi.acm.org/10.1145/1667048.1667050

## 1. INTRODUCTION

This article concerns techniques for static analysis of recursive programs that manipulate heap-allocated storage and perform destructive updating of pointer-valued fields. The goal is to recover shape descriptors that provide information about the characteristics of the data structures that a program's pointer variables can point to. Such information can be used to help programmers understand certain aspects of the program's behavior, to verify properties of the program, and to optimize or parallelize the program.

The work reported in the article builds on past work by several of the authors on static analysis based on 3-valued logic [Sagiv et al. 2002; Reps et al. 2003] and its implementation in the TVLA system [Lev-Ami and Sagiv 2000]. In this setting, two related logics come into play: an ordinary 2-valued logic, as well as a related 3-valued logic. A memory configuration, or store, is modeled by what logicians call a *logical structure*, which consists of a predicate (i.e., a relation of appropriate arity) for each predicate symbol of a *vocabulary* $\mathcal{P}$. A store is modeled by a 2-valued logical structure; a set of stores is abstracted by a (finite) set of bounded-size 3-valued logical structures. An individual of a 3-valued structure's universe either models a single memory cell or, in the case of a *summary individual*, a collection of memory cells.

The constraint of working with limited-size descriptors entails a loss of information about the store. Certain properties of concrete individuals are lost due to abstraction, which groups together multiple individuals into summary individuals: a property can be true for some concrete individuals of the group but false for other individuals. It is for this reason that 3-valued logic is used; uncertainty about a property's value is captured by means of the third truth value, 1/2.

One of the opportunities for scaling up this approach is to exploit the compositional structure of programs. In interprocedural dataflow analysis, one avenue for accomplishing this is to create a *summary transformer* for each procedure

$P$, and use the summary transformer at each call site at which $P$ is called. Each summary transformer must capture (an overapproximation of) the net effect of a call on $P$. To be able to create summary transformers, the abstract transformers for individual transitions must have a "composable representation"; that is, given the representations of two abstract transformers, it must be possible to represent their composition as an object of roughly the same size. One then carries out a fixpoint-finding procedure on a collection of equations in which each variable in the equation set has a transformer-valued value, that is, a value drawn from the domain of transformers, rather than a dataflow value proper.

A number of approaches to interprocedural dataflow analysis based on summary transformers are known [Cousot and Cousot 1977; Sharir and Pnueli 1981; Knoop and Steffen 1992; Reps et al. 1995, 2005; Sagiv et al. 1996]. However, not all program-analysis problems have abstract transformers that have a composable representation.

For some problems, it is possible to address this issue by working pointwise, tabulating composed transformers using either (i) sets of pairs that consist of an input abstract value and an output abstract value [Sharir and Pnueli 1981], or (ii) finer-granularity sets of pairs that capture how parts of an input abstract value influence parts of an output abstract value [Reps et al. 1995; Sagiv et al. 1996; Ball and Rajamani 2001]. In essence, these approaches start with the kinds of objects used in intraprocedural analysis and pair them together to create the objects that are used in interprocedural analysis.

However, for interprocedural shape analysis, tabulating pairs of 3-valued structures (the kinds of objects used in intraprocedural shape analysis) has significant drawbacks insofar as precision is concerned: in the 3-valued-logic approach to shape analysis, individuals, which model memory cells, do not have fixed identities; they are identified only up to their "distinguishing characteristics", namely, their values for a specific set of unary predicates. Because these "distinguishing characteristics" can change during the course of a procedure call, there is no way to identify individuals in an input abstract structure with their corresponding individuals in the output abstract structure. In essence, a pair of input/output 3-valued structures loses track of the correlations between the input and output values of an individual's unary predicates. Consequently, an approach based on tabulating composed transformers as sets of pairs of 3-valued structures provides only a weak characterization of a procedure's net effect, and is fundamentally limited in the properties that it can express.

All is not lost, however: instead of "abstracting and then pairing" (as discussed before), the solution is to "pair and then abstract."

*Observation* 1.1. By using a 3-valued structure over a doubled vocabulary $\mathcal{P} \uplus \mathcal{P}'$, where $\mathcal{P}' = \{p' \mid p \in \mathcal{P}\}$ and $\uplus$ denotes disjoint union, one can obtain a finite abstraction that relates the predicate values for an individual at the beginning of a transition to the predicate values for the individual at the end of the transition.

This approach provides a way to create much more accurate composable representations of transformers, and hence much more accurate summary transformers, for a broad class of problems. The advantages come from two effects.

- The addition of the second vocabulary changes the abstraction in use because individuals now have additional "distinguishing characteristics" [Sagiv et al. 2002].
- The second vocabulary helps permit the *changes* in a predicate to be tracked over a sequence of operations [Lev-Ami et al. 2000].

The benefit of these properties is that, in many cases, a relationship on the before and after values of a predicate can be tracked on individual locations or tuples of locations, over a sequence of operations, even when abstraction has been performed. The consequence is that two-vocabulary 3-valued structures provide more precise descriptors of relations between stores than an approach based on pairing abstract stores from an existing store abstraction.

Moreover, by extending the abstract domain of 3-valued logical structures with some new operations, it is possible to perform abstract interpretation of call and return statements without losing too much precision (see Section 6 and Section 7). We have used these ideas to create a context-sensitive shape-analysis algorithm for recursive programs that manipulate heap-allocated storage and perform destructive updating.

The "pair and then abstract" principle of Observation 1.1 is related to several well-known concepts.

*Pairing without abstraction.* The use of a doubled vocabulary is standard in logic-based reasoning about execution behavior: the transition relations of a language's concrete semantics are often expressed by means of formulas over present-state and next-state variables (e.g., Gries [1981], Manna and Pnueli [1995], and Clarke et al. [1999]). For instance, the semantics of a statement `x := y+1` can be expressed as the formula $(x' = y + 1) \wedge (y' = y)$. Similarly, a procedure's postcondition is often expressed using such a doubled vocabulary (i.e., the postcondition expresses a relation over input stores and output stores).

*Pairing and then numeric abstraction.* For analyzing programs that manipulate numeric data, a composable abstract transformer for a statement such as `x := y+1` can be created directly from the formula $(x' = y+1) \wedge (y' = y)$ when using the polyhedral abstract domain [Cousot and Halbwachs 1978]. The number of dimensions in each polyhedron used by the analyzer is double the number $|V|$ of numeric variables $V$ about which the analyzer is trying to obtain information. Each program variable has a primed and an unprimed version, and a polyhedron captures linear relations among the $2|V|$ variables.

In this article, we use Observation 1.1 to create composable abstract transformers for programs that manipulate nonnumeric data. Our work provides a new approach to performing context-sensitive interprocedural shape analysis, and allows us to verify properties of imperative programs with recursive procedure calls, heap-allocated storage, and destructive updating of pointer-valued fields.

The contributions of our work include the following.

(1) We introduce a new method for abstracting relations over memory configurations for use in abstract interpretation.

(2) We show how this method furnishes the elements needed for a compositional approach to shape analysis. In particular, abstracted relations are used to represent the shape transformation performed by a sequence of operations, and an overapproximation to relational composition can be performed using the meet operation of the domain of abstracted relations.

(3) We apply these ideas in a new algorithm for context-sensitive interprocedural shape analysis. The algorithm creates procedure summaries using abstracted relations over memory configurations, and the meet-based composition operation provides a way to apply the summary transformer for a procedure $P$ at each call site from which $P$ is called.

We have been able to apply this approach successfully to establish properties of both (i) recursive programs that manipulate lists, and (ii) recursive programs that manipulate binary trees. While list-manipulation programs can often be implemented in tail-recursive fashion, and hence can be converted easily into loop programs, tree-manipulation programs are much less easily converted to nonrecursive form. In particular, the shape properties that characterize sorted binary trees are complex and rely on global properties, whereas the shape properties that characterize sorted lists are mostly local properties (with cyclicity properties being the main exception).

*Organization.* The remainder of the article is organized as follows: Section 2 presents, at a semiformal level, several of the principles that lie behind our approach. Section 3 presents some background on 2-valued and 3-valued logic. Section 4 defines the language to which our analysis applies, and gives a concrete semantics, based on the use of 2-valued logical structures for representing memory configurations. Section 5 describes the abstraction of 2-valued logical structures with bounded-size 3-valued logical structures [Sagiv et al. 2002]. Our interprocedural shape analysis is based on a relational semantics, which establishes at each control point a relation between the input state of the enclosing procedure and the state at the current point. This semantics requires the ability to represent relations between memory configurations, which presents certain difficulties at the abstract level. Section 6 addresses this problem by abstracting relations between memory configurations using the same principles as those used to abstract sets of memory configurations in Section 5. Section 7 describes the interprocedural shape-analysis algorithm that we developed based on these ideas. Section 8 presents experimental results. Section 9 discusses related work.

## 2. OVERVIEW

In this section, we discuss at a semiformal level the “pairing” aspect of Observation 1.1 (“pair and then abstract”). Abstraction is the subject of Section 5. Section 7 applies the “pair and then abstract” principle in the context of interprocedural shape analysis.

```
typedef struct node {
  struct node *n;
  int data;
} *List;

List res;
void main(List l) {
  res = rev(l);
}
```

```
List rev(List x){
  List y, z;
  z = x->n;
  x->n = NULL;
  if (z != NULL){
    y = rev(z);
    z->n = x;
  }
  else y = x;
  return y;
}
```

Fig. 1. Recursive list-reversal program. The recursive function rev destructively reverses a nonempty, acyclic, singly-linked list using recursion to traverse the list.

```
(a)  [1]  a = <a 4-element list>; b = NULL; p = NULL;
     [2]  b = rev(a);
     [3]  p = b->n;
     [4]  .  .  .

(b)  [1]  a = <a 4-element list>; b = NULL; c = NULL;
     [2]  b = rev(a);
     [3]  c = rev(b);
     [4]  .  .  .
```

Fig. 2. Examples to illustrate one-vocabulary structures, two-vocabulary structures, transformer application, and procedure summaries.

Consider nonempty, acyclic, singly-linked lists constructed from nodes of the type List whose declaration is given in Figure 1. One of the issues discussed shortly concerns how to create a summary transformer for a procedure that reverses a list, using destructive updating. The summary transformer that we give applies both to recursive and nonrecursive destructive list-reversal procedures. Because summary transformers (also known as "procedure summaries") are particularly useful for analyzing recursive programs, the running example used in later sections of the article is the recursive list-reversal program shown in Figure 1. That procedure destructively reverses a nonempty, acyclic, singly-linked list using recursion to traverse the list.

In the remainder of this section, we discuss the two code fragments shown in Figure 2. Figure 3 depicts three four-element, singly-linked, acyclic lists. The nodes of each graph represent memory cells. An address-valued program variable ("pointer variable") that points to a given memory cell is represented by an arrow from the variable name to the node for the cell. (A pointer variable whose value is NULL is not shown.) The other arrows in the graph, labeled with n, represent the values of cells' n-fields. Figures 3(a), 3(b), and 3(c) represent lists that arise just before lines [2], [3], and [4] of Figure 2(a), respectively.

*Two kinds of pairing.* Figures 4 and 5 illustrate two different kinds of pairing operations that can be performed on lists.

—Figure 4(a) depicts a pair of one-vocabulary structures that represent the net transformation from just before line [2] of Figure 2(a) to just before line [3];

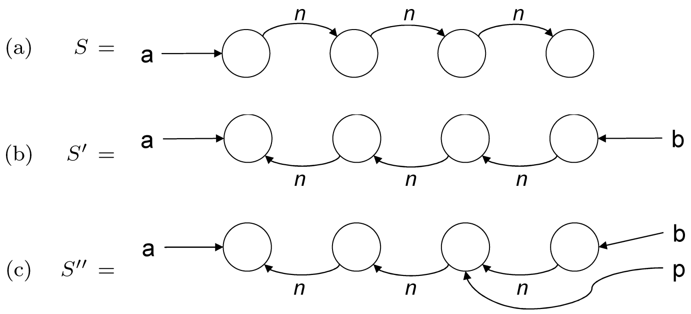


Fig. 3. (a) The (one-vocabulary) structure that represents a four-element acyclic list that is pointed to by a; (b) the (one-vocabulary) structure that represents the list from (a) after the operation "[2] b = rev(a);"; (c) the (one-vocabulary) structure that represents the list from (b) after the operation "[3] p = b->n;".

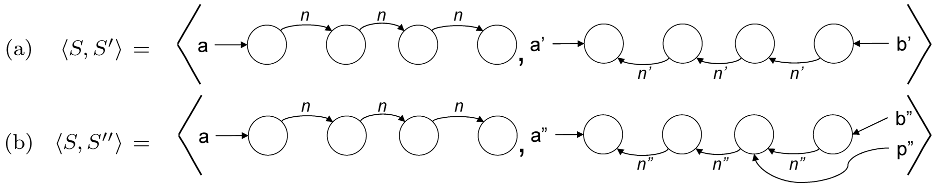


Fig. 4. Pairs of one-vocabulary structures that represent (a) the net transformation from just before line [2] of Figure 2(a) to just before line [3]; (b) the net transformation from just before line [2] of Figure 2(a) to just before line [4].

Figure 4(b) depicts a pair of one-vocabulary structures that represent the net transformation from just before line [2] of Figure 2(a) to just before line [4].

—Figure 5(a) depicts a two-vocabulary structure that represents the net transformation from just before line [2] of Figure 2(a) to just before line [3]; Figure 5(b) depicts a two-vocabulary structure that represents the net transformation from just before line [2] of Figure 2(a) to just before line [4].

A two-vocabulary structure has a single set of memory cells that are structured using two vocabularies. In Figure 5(a), one vocabulary is {a, b, p, n}; the second vocabulary is {a′, b′, p′, n′}. In Figure 5(b), the two vocabularies are {a, b, p, n} and {a″, b″, p″, n″}.[1] (In Figures (a) and 4(b), we have used single-primed and double-primed vocabularies in the respective second-component structures to emphasize how they correspond to the two-vocabulary structures of Figure 5(a) and 5(b). Strictly speaking, these should have been unprimed vocabularies.)

Even though we have drawn the list in the second component of the pair shown in Figure 4(a) so that each n′-edge appears to have been reversed from the n-edge in the first component, we have not given names to the nodes, and

[1] Variables b, p, and p′ do not appear in Figure 5(a) because they have the value NULL. Likewise, variables b and p do not appear in Figure 5(b) because they have the value NULL.

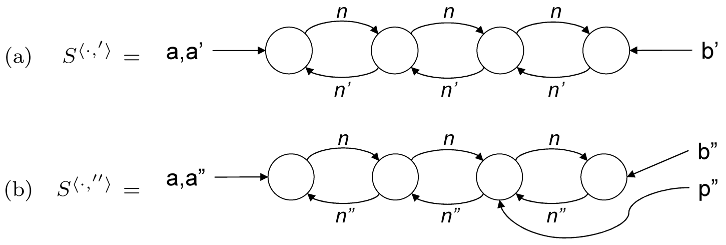

(a) $S^{\langle\cdot,'\rangle}$ =
a,a'
n
n
n
n'
n'
n'
b'
(b) $S^{\langle\cdot,''\rangle}$ =
a,a"
n
n
n
n"
n"
n"
b"
p"

Fig. 5. Two-vocabulary structures that represent (a) the net transformation from just before line [2] of Figure 2(a) to just before line [3]; (b) the net transformation from just before line [2] of Figure 2(a) to just before line [4]. (The superscript in each structure's name indicates what vocabularies are present in the structure; "·" stands for "unprimed".)

thus Figure 4(a) does not contain sufficient information to ensure that each of the original edges has, in fact, been reversed.[2]

In contrast, because there is a *unique* set of nodes in the two-vocabulary structure of Figure 5(a), we know that for each n-edge there is a corresponding reversed n′-edge, and vice versa.

*Transformer application.* Let $\tau$ denote the transformation produced by the statement "[3]p = b->n;" in line [3] of Figure 2(a). Consider three ways of depicting the effect.

—In terms of one-vocabulary structures, the transformation amounts to passing from Figure 3(b) to Figure 3(c).

$$\tau(S') = S''$$

—In terms of pairs of one-vocabulary structures, the transformation amounts to passing from Figure 4(a) to Figure 4(b).

$$\tau(\langle S, S'\rangle) = \langle S, \tau(S')\rangle = \langle S, S''\rangle$$

—In terms of two-vocabulary structures, the transformation amounts to passing from Figure 5(a) to Figure 5(b):

$$\tau(S^{\langle\cdot,'\rangle}) = S^{\langle\cdot,''\rangle},$$

where the superscript indicates what vocabularies are included in the structure ("·" stands for "unprimed").

*Two-vocabulary structures as procedure summaries.* Both (i) a pair of one-vocabulary structures, and (ii) a two-vocabulary structure provide a way to represent the net transformation performed by an operation (or a sequence of operations). However, as illustrated earlier, in the absence of indelible names for nodes, a two-vocabulary structure can represent information more precisely than a pair of one-vocabulary structures, and thus a two-vocabulary structure

[2] Although it would be easy to give indelible names to nodes in each concrete list, it will become apparent in Section 5 that this is not the case for nodes in abstract lists. The discussion in this section is intended to convey, using concrete lists, how we overcome the lack of indelible names for nodes in abstract lists.

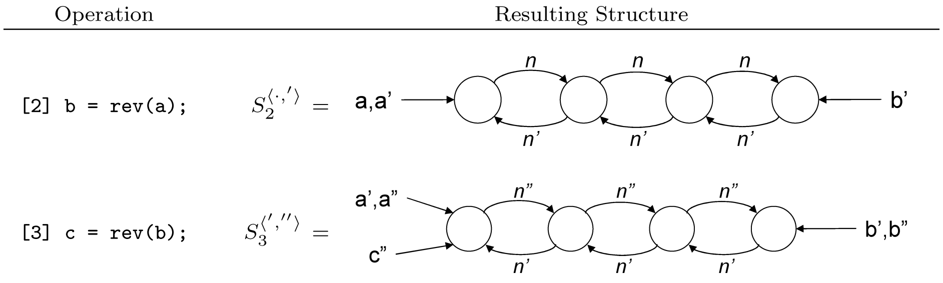


Fig. 6. The two-vocabulary structures that summarize (a) the transformation performed by "[2] b = rev(a);", and (b) the transformation performed by "[3] c = rev(b);".

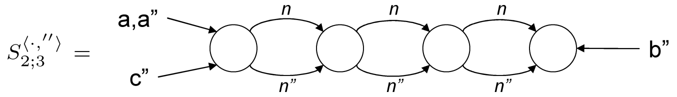


Fig. 7. The two-vocabulary structure $S_{2;3}^{\langle \cdot, '' \rangle}$ represents the net transformation performed by the sequence "[2]b = rev(a); [3]c = rev(b)". Note that for each (unprimed) n-edge there is a corresponding (double-primed) n″-edge, and vice versa.

can provide a more precise procedure summary than a pair of one-vocabulary structures.

In the remainder of this section, we discuss the code fragment shown in Figure 2(b). Structure $S_2^{\langle \cdot, ' \rangle}$ in Figure 6(a) summarizes the transformation performed by "[2] b = rev(a);", and structure $S_3^{\langle ', '' \rangle}$ in Figure 6(b) summarizes the transformation performed by "[3] c = rev(b);".

*Transformer composition.* The result of composing the transformations represented by two two-vocabulary structures can be expressed as another two-vocabulary structure. For instance, consider the two-vocabulary structure $S_{2;3}^{\langle \cdot, '' \rangle}$ shown in Figure 7, which represents the result of composing Figure 6(b) with Figure 6(a) to obtain a two-vocabulary structure for the sequence "[2] b = rev(a); [3] c = rev(b);".

The composition of the transformations represented by two two-vocabulary structures can be expressed in terms of a meet operation on three-vocabulary structures. To explain this, we introduce the graphical notation of dotted edges to represent unknown information (i.e., with truth value 1/2). For instance, Figure 8(a) and Figure 8(b) show two three-vocabulary structures $S_2^{\langle \cdot, ', 1/2'' \rangle}$ and $S_3^{\langle 1/2, ', '' \rangle}$, respectively, where the symbol 1/2 in the superscript of a structure name indicates that the structure has only unknown information for a given vocabulary. Note that $S_2^{\langle \cdot, ', 1/2'' \rangle}$ and $S_3^{\langle 1/2, ', '' \rangle}$ are three-vocabulary structures that correspond to the two-vocabulary structures $S_2^{\langle \cdot, ' \rangle}$ and $S_3^{\langle ', '' \rangle}$ from Figure 6, respectively.

We introduce the *meet* operation ($\sqcap$), where "unknown" $\sqcap$ "definite information" yields "definite information".[3] With this notation, the composition

[3] "Definite information" means "definitely present" (*true*, denoted by 1) or "definitely absent" (*false*, denoted by 0). Thus, $1/2 \sqcap 1 = 1 = 1 \sqcap 1/2$ and $1/2 \sqcap 0 = 0 = 0 \sqcap 1/2$.

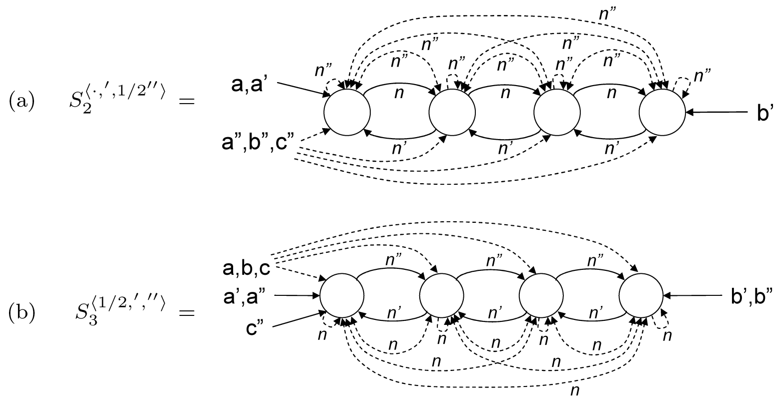

(a) $S_2^{\langle\cdot,',1/2''\rangle}$ =
a,a'
a",b",c"
b'
n
n'
n"
(b) $S_3^{\langle 1/2,',''\rangle}$ =
a,b,c
a',a"
c"
b',b"
n
n'
n"

Fig. 8. Three-vocabulary structures for the two-vocabulary structures from Figure 6. Dotted edges indicate predicate tuples that have the value 1/2 (and hence correspond to information that is unknown). In (a), the unprimed and single-primed vocabularies capture the transformation performed by [2] `b = rev(a);`, and the information in the double-primed vocabulary (predicates a″, b″, c″, and n″) is unknown. In (b), the single-primed and double-primed vocabularies capture the transformation performed by [3] `c = rev(b);`, and the information in the unprimed vocabulary (predicates a, b, c, and n) is unknown.

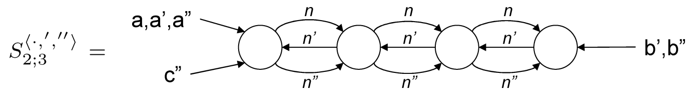


Fig. 9. The three-vocabulary structure $S_{2;3}^{\langle\cdot,',''\rangle}$ obtained from the meet ($\sqcap$) of the two structures from Figure 8: $S_3^{\langle 1/2,',''\rangle} \sqcap S_2^{\langle\cdot,',1/2''\rangle}$. Note that for each (unprimed) n-edge there is a corresponding (double-primed) n″-edge, and vice versa.

$S_3^{\langle ',''\rangle} \circ S_2^{\langle\cdot,'\rangle}$ of the transformations represented by two two-vocabulary structures $S_2^{\langle\cdot,'\rangle}$ and $S_3^{\langle ',''\rangle}$ can be expressed in terms of three-vocabulary structures as

$$\begin{aligned} S_3^{\langle ',''\rangle} \circ S_2^{\langle\cdot,'\rangle} &= \text{project}_{1,3}\big(S_3^{\langle 1/2,',''\rangle} \sqcap S_2^{\langle\cdot,',1/2''\rangle}\big) \\ &= S_{2;3}^{\langle\cdot,''\rangle}. \end{aligned}$$

The three-vocabulary structure $S_{2;3}^{\langle\cdot,',''\rangle}$ obtained from $S_3^{\langle 1/2,',''\rangle} \sqcap S_2^{\langle\cdot,',1/2''\rangle}$ is shown in Figure 9. Finally, by projecting away the “middle” (single-primed) vocabulary from $S_{2;3}^{\langle\cdot,',''\rangle}$, we obtain the two-vocabulary composition result $S_{2;3}^{\langle\cdot,''\rangle}$ shown in Figure 7.

*How these ideas are used in relational shape analysis.* In Section 5, we introduce a way to use 3-valued structures as abstractions of sets of 2-valued structures. In Section 6, this is extended to using two-vocabulary 3-valued structures as abstractions of transformations on 2-valued structures. This provides what is needed for a compositional approach to shape analysis:

—the 3-valued analog of the two-vocabulary version of transformer application can be used for intraprocedural propagation;

—the 3-valued analog of transformer composition can be used for interprocedural propagation.

Two-vocabulary 3-valued structures are used as summary transformers for the shape transformations performed by the possible sequences of operations in each procedure, and an overapproximation of composition can be performed using the meet operation on three-vocabulary 3-valued structures. In particular, it is possible to perform an overapproximation of the composition of the transformations represented by two two-vocabulary 3-valued structures, $(S^{\#})^{\langle \cdot,' \rangle}$ and $(T^{\#})^{\langle ',''\rangle}$, by (i) promoting them to three-vocabulary 3-valued structures $(S^{\#})^{\langle \cdot,',1/2''\rangle}$ and $(T^{\#})^{\langle 1/2,',''\rangle}$, (ii) taking their meet, and (iii) projecting away the middle vocabulary. (See Section 6.5.)

## 3. PRELIMINARIES

### 3.1 2-Valued First-Order Logic

We briefly discuss definitions related to first-order logic. We assume a *vocabulary* $\mathcal{P}$ of predicate symbols and a set of variables, usually denoted by $v, v_1, \ldots$. Formulas are defined by the following syntax.

$$
\begin{array}{rll}
\varphi \ ::= & \mathbf{1} & \text{logical literal} \\
\mid & p(v_1, \ldots, v_k) & \text{where } p \text{ is a predicate symbol of arity } k \\
\mid & \neg\varphi \mid \varphi \vee \varphi & \text{logical connectives} \\
\mid & \exists v : \varphi & \text{existential quantification}
\end{array}
\tag{1}
$$

For reasons that will be made explicit in the next paragraph, we do not include the formula $v_1 = v_2$ in the grammar itself. Instead, we assume that the vocabulary $\mathcal{P}$ contains a special predicate symbol *eq* of arity 2 that will have a special interpretation. We will write $v_1 = v_2$ and $v_1 \neq v_2$ for $eq(v_1, v_2)$ and $\neg eq(v_1, v_2)$. The literal $\mathbf{0}$, the connectives $\Rightarrow$ and $\wedge$, and the quantifier $\forall v$ are defined in the usual way, in terms of items in grammar (1). A conditional expression $\varphi_1 ? \varphi_2 : \varphi_3$ is an abbreviation for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3)$. The notion of free variables is defined in the usual way.

The set $\{0, 1\}$ of (2-valued) truth values is denoted by $\mathbb{B}$. A *2-valued logical structure* $S = (U, \iota)$ is a pair, where the *universe* $U$ is a set of *individuals* and the *valuation* $\iota : \mathcal{P} \to \bigcup_{k \geq 0}(U^k \to \mathbb{B})$ maps each predicate symbol of arity $k$ to a predicate (or truth-valued function). The set of 2-valued structures over a *vocabulary* $\mathcal{P}$ is denoted by $2\text{−STRUCT}[\mathcal{P}]$. We assume that for any $(U, \iota) \in 2\text{−STRUCT}[\mathcal{P}]$, $\iota(eq)$ is defined by $\iota(eq)(u_1, u_2) = (u_1 = u_2)$.

An *assignment* $Z : \{v_1, \ldots, v_k\} \to U$ maps free variables (implicitly with respect to a formula) to individuals. Given a 2-valued logical structure $S = (U, \iota)$ and an assignment $Z$ of free variables, the (2-valued) meaning of a formula $\varphi$, denoted by $[\![\varphi]\!]^S(Z)$, is defined in Table I by induction on the syntax of $\varphi$. A logical structure *satisfies* a closed formula $\varphi$ (i.e., without free variables), denoted by $S \models \varphi$, iff $[\![\varphi]\!]^S = 1$. For open formulas, satisfaction with respect to assignment $Z$ is defined by $S, Z \models \varphi$, iff $[\![\varphi]\!]^S(Z) = 1$.

Table I. Meaning of First-Order Formulas, Given a Logical Structure $S = (U, \iota)$ and an Assignment $Z$

$$\begin{gathered}
\llbracket \mathbf{1} \rrbracket^S(Z) = 1 \\
\llbracket p(v_1, \dots, v_k) \rrbracket^S(Z) = \iota(p)(Z(v_1), \dots, Z(v_k)) \\
\llbracket \neg\varphi_1 \rrbracket^S(Z) = 1 - \llbracket \varphi_1 \rrbracket^S(Z) \\
\llbracket \varphi_1 \vee \varphi \rrbracket^S(Z) = \max(\llbracket \varphi_1 \rrbracket^S(Z), \llbracket \varphi_1 \rrbracket^S(Z)) \\
\llbracket \exists v_1 : \varphi_1 \rrbracket^S(Z) = \max_{u \in U} \llbracket \varphi_1 \rrbracket(Z[v_1 \mapsto u])
\end{gathered}$$

## 3.2 3-Valued First-Order Logic

We now extend the definitions from Section 3.1 to 3-valued logic, in which a third truth value, denoted by 1/2, represents uncertainty. The set $\mathbb{B} \cup \{1/2\}$ of 3-valued truth values is denoted by $\mathbb{T}$, and is partially ordered by the order $l \sqsubset 1/2$ for $l \in \mathbb{B}$.

A 3-*valued logical structure* $S = (U, \iota)$ is almost identical to a 2-valued structure, except for the fact that $\iota : \mathcal{P} \to \bigcup_{k \geq 0}(U^k \to \mathbb{T})$ maps each predicate symbol of arity $k$ to a 3-valued truth-valued function. The syntax of formulas defined in Eq. (1) is extended with the logical literal $\mathbf{1/2}$, which is given the meaning $\llbracket \mathbf{1/2} \rrbracket^S = 1/2$. The meaning of other syntactic constructs is still defined by Table I. Note that the operations "−" and "max" can accept the value 1/2 as an operand.

A 3-valued logical structure *potentially satisfies* a closed (3-valued) formula $\varphi$, denoted by $S \models \varphi$, iff $\llbracket \varphi \rrbracket^S \in \{1/2, 1\}$. For open formulas, we have $S, Z \models \varphi$, iff $\llbracket \varphi \rrbracket^S(Z) \in \{1/2, 1\}$.

We refer to Sagiv et al. [2002] for the extension of first-order 2- and 3-valued logic with transitive closure, which we have omitted here for the sake of simplicity. The transitive closure of a formula with two free variables $\varphi(v_1, v_2)$ is denoted by $\varphi^*(v_1, v_2)$.

*Embedding of* 3-*valued logical structures.* To abstract memory configurations represented by logical structures, we use the following notion of embedding.

*Definition* 3.1. Given $S = (U, \iota)$ and $S' = (U', \iota')$, two 3-valued structures over the same vocabulary $\mathcal{P}$, and $f : U \to U'$, a surjective function, $f$ embeds $S$ in $S'$, denoted by $S \sqsubseteq^f S'$, if for all $p \in \mathcal{P}$ and $u_1, \dots, u_k \in U$,

$$\iota(p)(u_1, \dots, u_k) \sqsubseteq \iota'(p)(f(u_1), \dots, f(u_k)).$$

If, in addition,

$$\iota'(p)(u'_1, \dots, u'_k) = \bigsqcup_{u_1 \in f^{-1}(u'_1), \dots, u_k \in f^{-1}(u'_k)} \iota(p)(u_1, \dots, u_k)$$

then $S'$ is the tight embedding of $S$ with respect to $f$, denoted by $S' = f(S)$.

Intuitively, $f(S)$ is obtained by merging individuals of $S$ and by defining accordingly the valuation of predicates (in the most precise way). Observe that $\sqsubseteq^{\text{id}}$, which will be denoted simply by $\sqsubseteq$, is the natural information order between structures that share the same universe. Note that one has $S \sqsubseteq^f S' \Leftrightarrow f(S) \sqsubseteq^{\text{id}} S'$.

We can now explain the usefulness of the *eq* predicate. Let $S = (U, \iota) \in$ 2−STRUCT and $S' = (U', \iota') = f(S)$. We have

$$\iota'(eq)(u'_1, u'_2) = \begin{cases} 1 & \text{if } \forall u_1 \in f^{-1}(u'_1), \forall u_2 \in f^{-1}(u'_2) \ : \ \iota(eq)(u_1, u_2) = 1 \\ 0 & \text{if } \forall u_1 \in f^{-1}(u'_1), \forall u_1 \in f^{-1}(u'_2) \ : \ \iota(eq)(u_1, u_2) = 0 \\ 1/2 & \text{otherwise} \end{cases}$$

which can be simplified to the following.

$$\iota'(eq)(u'_1, u'_2) = \begin{cases} 1 & \text{if } u'_1 = u'_2 \wedge |f^{-1}(u'_1)| = 1 \\ 0 & \text{if } u'_1 \neq u'_2 \\ 1/2 & \text{if } u'_1 = u'_2 \wedge |f^{-1}(u'_1)| > 1 \end{cases}$$

Note that $u'_1 = u'_2$ in the simplified definition is not a shorthand for $eq(u'_1, u'_2)$; it evaluates to true whenever $u'_1$ and $u'_2$ are the same individual of $U'$. Similarly, $u'_1 \neq u'_2$ evaluates to true when $u'_1$ and $u'_2$ are distinct individuals of $U'$. Hence, for any $S'' = (U'', \iota'') \sqsupseteq^f S$, if for some $u'' \in U''$ $\iota''(eq)(u'', u'') = 1$, then $|f^{-1}(u'')| = 1$, otherwise $|f^{-1}(u'')| \geq 1$. Consequently, the value of the formula $eq(v, v)$ evaluated in a 3-valued structure $S''$ indicates whether an individual of $S''$ represents exactly one individual in each of the structures $S$ that can be embedded into $S''$, or at least one individual.

The following preservation theorem about the interpretation of logical formulas allows to interpret logical formulas in embedded structures in a conservative way with respect to the original structure.

THEOREM 3.2 (EMBEDDING THEOREM [SAGIV ET AL. 2002]). *Let $S = (U, \iota)$ and $S' = (U', \iota')$ be two 3-valued structures, such that there exists an embedding function $f$ with $S \sqsubseteq^f S'$. Then, for any formula $\varphi(v_1, \ldots, v_k)$ and assignment $Z : \{v_1, \ldots, v_k\} \to U$ of free variables of $\varphi$, we have*

$$[\![\varphi]\!]_3^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S'}(Z'),$$

*where $Z' : \{v_1, \ldots, v_k\} \to U'$ is the abstract assignment defined by $Z'(v_i) = f(Z(v_i))$.*

## 4. PROGRAMS AND MEMORY CONFIGURATIONS

We consider programs written in an imperative programming language in which

(1) it is forbidden to take the address of a local variable, a global variable, a parameter, or a function;
(2) parameters are passed by value;
(3) pointer arithmetic is forbidden.

These restrictions prevent direct aliasing among variables; thus, only nodes in heap-allocated structures can be aliased. The third feature makes memory

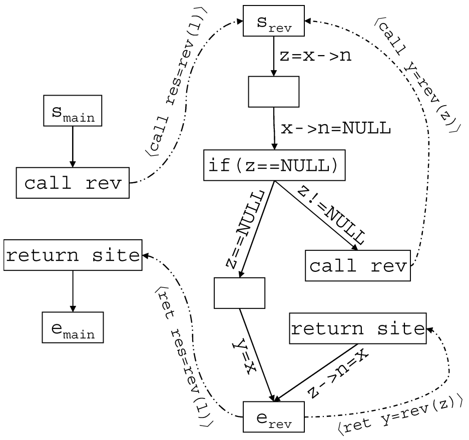


Fig. 10. Interprocedural CFG of the list-reversal program.

configurations invariant under permutations of addresses. Note that both Java and Ml follow these conventions.

## 4.1 Program Syntax

A *program* is defined by a set of procedures $P_i$, $0 \leq i \leq K$. Each procedure has local variables, formal *input parameters*, and *output parameters*. To simplify our notation, we will assume that each procedure has only one input parameter and one output parameter; the generalization to multiple parameters is straightforward. We also assume that an input parameter is not modified during the execution of the procedure. This assumption is made solely for convenience, and involves no loss of generality because it is always possible to copy input parameters to additional local variables.

Thus, a *procedure* $P_i = \langle \mathsf{fpi}_i, \mathsf{fpo}_i, \mathcal{L}_i, G_i \rangle$ is defined by its input parameter $\mathsf{fpi}_i$, its output parameter $\mathsf{fpo}_i$, its set of local variables $\mathcal{L}_i$ (containing $\mathsf{fpi}_i$ and $\mathsf{fpo}_i$), and $G_i$, its intraprocedural Control Flow Graph (CFG).

A program is represented by a directed graph $G^* = (N^*, E^*)$, called an *interprocedural CFG*. $G^*$ consists of a collection of intraprocedural CFGs $G_1, G_2, \ldots, G_K$, one of which, $G_{main}$, represents the program's main procedure. Each CFG $G_i$ contains exactly one *start* node $s_i$ and exactly one *exit* node $e_i$. The nodes of a CFG represent control points and its edges represent individual statements and branches of a procedure in the usual way. A procedure call statement relates a *call* node and a *return-site* node. For $n \in N^*$, $proc(n)$ denotes the (index of the) procedure that contains $n$. In addition to the ordinary intraprocedural edges that connect the nodes of the individual flowgraphs in $G^*$, each procedure call, represented by call-node $c$ and return-site node $r$, has two edges: (1) a *call-to-start* edge from $c$ to the start node of the called procedure; (2) an *exit-to-return-site* edge from the exit node of the called procedure to $r$. The functions *call* and *ret* record matching call and return-site nodes: $call(r) = c$ and $ret(c) = r$. We assume that a start node has no incoming edges except call-to-start edges.

Table II. Two Related Models of a Program State, Where $D$ May be $\mathbb{B}$ or `Int`

Set-theoretic view

| Set of cells<br>Cell | Pointer variable `z`<br>$z \in \mathsf{Cell} \cup \{\mathsf{NULL}\}$ | Pointer field `n`<br>$n \in \mathsf{Cell} \to \mathsf{Cell} \cup \{\mathsf{NULL}\}$ | Data variable `x`<br>$x \in D$ | Data field `d`<br>$d \in \mathsf{Cell} \to D$ |
|---|---|---|---|---|
| $U$<br>Universe | $z : U \to \mathbb{B}$<br>Unary relation | $n : U \times U \to \mathbb{B}$<br>Binary relation | $x : D$<br>Nullary function | $d : U \to D$<br>Unary function |

Logical view

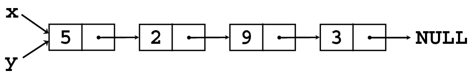


Fig. 11. A possible store, consisting of a four-node linked list pointed to by `x` and `y`.

## 4.2 Representing Memory Configurations

Consider a program that consists of several procedures, and, for the moment, ignore the stack of activation records in each state. At a given control point, a program state $s \in State$ is defined by the values of the local variables and the heap. We describe two ways in which such a state $s$ can be modeled (see Table II).

- The set-theoretic model is perhaps more intuitive. We consider a fixed set $\mathsf{Cell}$ of memory cells. The value of a pointer variable `z` is modeled by an element $z \in \mathsf{Cell} \cup \{\mathsf{NULL}\}$, where $\mathsf{NULL}$ denotes the null value. If cells have a pointer-valued field `n`, the values of `n` fields are modeled by a function $n : \mathsf{Cell} \to \mathsf{Cell} \cup \{\mathsf{NULL}\}$ that associates with each memory cell the cell pointed to by the field. If cells have an `Int`-valued (or, more generally, a data-valued) field `x`, the values of `x` fields are modeled by a function $d : \mathsf{Cell} \to \mathtt{Int}$ that associates with each memory cell the value of the corresponding field.
- Sagiv et al. [2002] model states using the tools of logic (refer to Section 3.1). Each state is modeled as a 2-valued logical structure: the set of memory cells is replaced by a universe $U$ of individuals; the value of a program variable `z` is defined by a unary predicate on $U$; and the value of a field `n` is defined by a binary predicate on $U$. Integrity constraints are used to capture the fact that, for instance, a unary predicate $z$ that represents what program variable `z` points to can have the value "true" for at most one memory cell [Sagiv et al. 2002].

We use the term "predicate of arity $n$" for a Boolean function $U^n \to \mathbb{B}$. We use $\mathcal{P}_n$ to denote the set of predicates symbols of arity $n$, and $\mathcal{N}$ to denote the set of integer-valued function symbols. With such notation, the concrete state-space considered is[4]

$$State = (U \to \mathbb{B})^{|\mathcal{P}_1|} \times (U^2 \to \mathbb{B})^{|\mathcal{P}_2|} \times (U \to \mathtt{Int})^{|\mathcal{N}|}, \qquad (2)$$

[4] Eq. (2) is the concrete state-space that one has when the techniques of Sagiv et al. [2002] are combined with those of Gopan et al. [2004]. To simplify Eq. (2), we have omitted nullary predicates, which would be used to model Boolean-valued variables, and nullary functions, which would be used to model data-valued variables.

where $|E|$ denotes the size of a finite set $E$. A concrete property in $\wp(State)$ is thus a set of tuples, each field of which is a function.

From now on, for the sake of simplicity, we will first perform the trivial abstraction of the concrete state space defined by Eq. (2) to the state-space

$$State = (U \rightarrow \mathbb{B})^{|\mathcal{P}_1|} \times (U^2 \rightarrow \mathbb{B})^{|\mathcal{P}_2|} \tag{3}$$

In this case, a state $S \in State$ can be represented by a 2-valued logical structure $(U, \iota)$ (Section 3.1), where the valuation function $\iota : \mathcal{P} \rightarrow \bigcup_k (U^k \rightarrow \mathbb{B})$ associates each predicate symbol of arity $k$ with a $k$-ary relation over $U$. We thus have $State \simeq 2\mathrm{-STRUCT}[\mathcal{P}]$.

In the sequel, we also assume that the universe $U$ is infinite. Because all infinite countable sets are isomorphic, we can omit the universe in declarations of 2-valued structures $S = (U, \iota) \in 2\mathrm{-STRUCT}[\mathcal{P}]$, so that $S$ will denote both the 2-valued structure and its valuation function $\iota$.

*Remark* 4.1. Because we want shape properties to be invariant under permutations of memory cells, we implicitly quotient State by the equivalence relation $S \approx S'$ if there is a permutation $f : U \rightarrow U$ such that

$$\forall p \in \mathcal{P} : S'(p)(u_1, \ldots, u_k) = S(p)(f(u_1), \ldots, f(u_k)).$$

The predicates that are part of the underlying semantics of the language to be analyzed are called *core predicates*. They will be distinguished from additional predicates that will be introduced later when abstracting concrete heaps. The set of core predicates that are used is dictated by the semantics of the programming language to be analyzed. (The programming language can have a degree of abstraction already built into it by the analysis designer, as illustrated by Remarks 4.2 and 4.3 that follow). For the programs that we consider, and the part of the state-space that we chose to analyze (Eq. (3)), we need to introduce a core predicate for each program variable and data-structure field, following Table II. The set of core predicates is thus uniquely defined for a given program.

*Remark* 4.2 (*Modeling Dynamic Memory Allocation*). The free memory pool required for dynamic memory allocation and deallocation is modeled using a core predicate *free*$(v)$, which has the value true for the unbounded number of nodes modeling free memory cells.

*Remark* 4.3 (*Modeling Ordering Among Cells' Data Values*). In some experiments of Section 8.2, we model lists and trees that are ordered with respect to integer keys. However, according to Eq. (3), we abstract integer values and we cannot compare such keys directly. Instead, we introduce a special core predicate *leq*$(v_1, v_2)$, which (i) is a total order, and (ii) has the value true on $(v_1, v_2)$ whenever the key of cell $v_1$ is less than or equal to the key of cell $v_2$. This core predicate can be seen as an abstraction of the predicate `cell1->key <= cell2->key` when the state-space of Eq. (2) is abstracted into the state-space of Eq. (3).

### 4.3 Semantics of Intraprocedural Operations

The usefulness of adopting the logical view for modeling memory becomes apparent when defining the semantics of instructions. This is because one can use the language of first-order logic for specifying how predicates (and hence logical structures and memory configurations) are transformed by the program's operations.

In this section, we only discuss intraprocedural operations; the problem of defining the semantics of interprocedural operations is left to Section 7.1.

Generally speaking, the concrete operational semantics of a programming language is defined by specifying a state transformer for each kind of operation associated with intraprocedural edges of the control-flow graph. We distinguish among the operations *statements*, which modify the program state, from *conditions*, which select program states that satisfy the conditions. The semantics of a statement $\mathsf{stm}$ is a transformer with signature $[\![\mathsf{stm}]\!] : \mathit{State} \rightarrow \mathit{State}$; the semantics of a condition $\mathsf{cond}$ is a predicate $[\![\mathsf{cond}]\!] : \mathit{State} \rightarrow \mathbb{B}$, which can be lifted to a transformer with signature $[\![\mathsf{cond}]\!] : \wp(\mathit{State}) \rightarrow \wp(\mathit{State})$ that filters out the states not satisfying the condition.

4.3.1 *Statements.* The transformer of a statement $\mathsf{stm}$ acts on states modeled as logical structures. It is defined using a collection of *predicate-update formulas*, $c(v_1, \ldots, v_k) = \varphi^c_{\mathsf{stm}}(v_1, \ldots, v_k)$, one for each core predicate $c$ (see Sagiv et al. [2002]). These formulas define how the core predicates of a logical structure $S$ are transformed by the statement `stm` to create a logical structure $S'$; they define the value of predicate $c$ in $S'$ as a function of $c$'s value in $S$. Formally,

$$\begin{array}{lcl} [\![\mathsf{stm}]\!] : \mathit{State} & \longrightarrow & \mathit{State} \\ \quad S & \longmapsto & S' \\ \text{where} & & \\ \multicolumn{3}{l}{\forall c \in \mathcal{P} : S'(c)(u_1, \ldots, u_k) = [\![\varphi^c_{\mathsf{stm}}(v_1, \ldots, v_k)]\!]^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]) \, .} \end{array} \tag{4}$$

For instance, the semantics of the assignment statement `z->n = NULL;` is specified by the predicate-update formulas

$$\varphi^n_{\mathsf{stm}}(v_1, v_2) = n(v_1, v_2) \wedge \neg z(v_1), \qquad \varphi^c_{\mathsf{stm}}(v_1, \ldots, v_k) = c(v_1, \ldots, v_k) \ \text{ for } c \neq n.$$

The predicate-update formula $\varphi^n_{\mathsf{stm}}$ should be read as follows: "If the cell $v_1$ is not pointed to by the variable $z$, leave the $n$ field of the cell $v_1$ unchanged, otherwise assign it the value $\mathsf{NULL}$ (represented by $n(v_1, v_2) = \text{false}$ for every cell $v_2$)." We assume that the statements of the analyzed program are decomposed into the elementary statements listed in Table III (which is always possible for the class of languages considered in this article). The elementary statements modify the value of at most one core predicate. We omit writing explicit predicate-update formulas for predicates that are unchanged by a statement. (The omitted formulas merely express the identity transformation.)

4.3.2 *Conditions.* The semantics of a condition $\mathsf{cond}$ is defined by a *precondition formula* $\varphi_{\mathsf{cond}}$, which is a nullary formula that filters out structures that should not follow the transition along edges $e$ labeled by the condition.

Table III. Predicate-Update Formulas for Statements

| Statement | Predicate-update Formula |
|---|---|
| `z = NULL` | $\iota^z(v) = \mathbf{0}$ |
| `z = y` | $\iota^z(v) = y(v)$ |
| `z = y->sel` | $\iota^z(v) = \exists v_1 : y(v_1) \wedge sel(v_1, v)$ |
| `z->sel = NULL` | $\iota^{sel}(v_1, v_2) = sel(v_1, v_2) \wedge \neg z(v_1)$ |
| `z->sel = y (assuming that z->sel = NULL)` | $\iota^{sel}(v_1, v_2) = sel(v_1, v_2) \vee (z(v_1) \wedge y(v_2))$ |

Table IV. Precondition Formulas for Conditions

| Condition | Precondition formula |
|---|---|
| `z == NULL` | $\forall v : \neg z(v)$ |
| `z != NULL` | $\exists v : z(v)$ |
| `z1 == z2` | $\forall v : z1(v) \Leftrightarrow z2(v)$ |
| `z1 != z2` | $\exists v : \neg(z1(v) \Leftrightarrow z2(v))$ |
| `z->sel == NULL (assuming that z != NULL)` | $\forall v_1, v_2 : z(v_1) \Rightarrow \neg sel(v_1, v_2)$ |
| `z->sel != NULL` | $\exists v_1, v_2 : z(v_1) \wedge sel(v_1, v_2))$ |
| `z1->sel == z2 (assuming that z1 != NULL)` | $\forall v_1, v_2 : z1(v_1) \Rightarrow (sel(v_1, v_2) \Leftrightarrow z2(v_2))$ |
| `z1->sel != z2` | $\exists v_1, v_2 : z1(v_1) \wedge \neg(sel(v_1, v_2) \Leftrightarrow z2(v_2))$ |

Formally,

$$\begin{array}{rccl} [\![\varphi_{\mathsf{cond}}]\!] : & \wp(State) & \longrightarrow & \wp(State) \\ & X & \longmapsto & X' \subseteq X \\ & \text{where} & & \\ & \multicolumn{3}{l}{X' = \{S \in X \mid S \models \varphi_{\mathsf{cond}}\} .} \end{array} \tag{5}$$

For instance, the semantics of the condition `z->n != NULL` is given by the precondition formula

$$\exists v_1, v_2 : z(v_1) \wedge n(v_1, v_2),$$

which evaluates to false on logical structures for which the $n$ field of the cell pointed to by `z` (if any) is equal to NULL. Table IV gives the complete semantics of conditions. Program assumptions, such as `z!=NULL` at the point of a dereference of `z`, are checked by the analysis using the "halt" instruction of the TVLA system [Lev-Ami and Sagiv 2000], which generates an alert when a program assumption is not satisfied.

4.3.3 *Memory Allocation and Deallocation.* Remark 4.2 introduced the predicate $free(v)$ for modeling the free memory pool. The semantics of a memory deallocation instruction `dealloc(z)` is defined using the predicate-update formulas $\tau^z(v) = \mathbf{0}$ and $\tau^{free}(v) = free(v) \vee z(v)$. Intuitively, the semantics of a memory allocation instruction `z = alloc()` is to pick randomly a node $v_0$ with $free(v_0) = \mathbf{1}$, and then update $free(v)$ and $z(v)$ using predicate-update formulas $\tau^{free}(v) = free(v) \wedge \neg eq(v, v_0)$ and $\tau^z(v) = eq(v, v_0)$.[5]

[5] Unfortunately, "picking a node randomly" cannot be easily expressed in 2-valued logic, so we define it directly in 3-valued logic using the special operator Focus that will be introduced in Section 5. (To conserve space, we do not give the precise definition here.) An alternative would have been to employ a concrete model of the free memory pool, for example, using a singly-linked list, but this would have increased the complexity of the summaries of procedures that perform allocation and deallocation.

## 5. ABSTRACTING MEMORY CONFIGURATIONS

In this section, we discuss the abstraction method developed by Sagiv et al. [2002], which maps 2-valued logical structures (of arbitrary size) to 3-valued logical structures of bounded size.

The problem with representing and manipulating 2-valued structures is the unbounded universe $U$. Consequently, the starting point for abstracting a 2-valued structure is the abstraction of the universe $U$ to an abstract universe $U^\sharp$ of bounded size. Intuitively, the abstraction consists of (i) merging concrete individuals into a bounded number of abstract individuals $U^\sharp$, and (ii) replacing the concrete predicates by abstract versions in which the values of the tuples reflect how concrete individuals have been merged to create the abstract individuals.

### 5.1 The Abstraction Principle

Given a finite set $U^\sharp$ with a surjective function $f : U \to U^\sharp$, one can define the following Galois connection, using the tight embedding on logical structures induced by $f$ and the partial order defined on 3-valued structures (see Definition 3.1).

$$\begin{array}{rcl} \wp(2\text{−STRUCT}) & \underset{\gamma_f}{\overset{\alpha_f}{\leftrightarrows}} & 3\text{−STRUCT} \\ \alpha_f(X) & = & \bigsqcup\limits_{S \in X} f(S) \\ \gamma_f(S^\sharp) & = & \{S \mid S \sqsubseteq^f S^\sharp\} \end{array}$$

In this abstraction, sets of valuations for predicate symbols $\iota : \mathcal{P} \to \left(\bigcup_k U^k \to \mathbb{B}\right)$ are abstracted with a single abstract valuation $\iota : \mathcal{P} \to \left(\bigcup_k (U^\sharp)^k \to \mathbb{T}\right)$.

### 5.2 The Abstract Domain of 3-Valued Structures

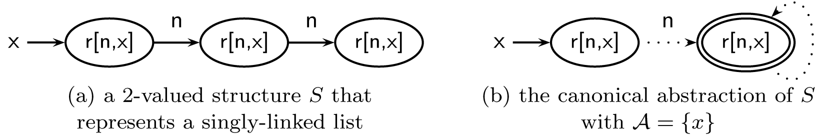
The abstraction principle depicted before is parameterized by a finite abstraction of the universe $U$ of 2-valued structures. The idea behind *canonical abstraction* [Sagiv et al. 2002] is to choose a subset $\mathcal{A} \subseteq \mathcal{P}_1$ of abstraction predicates and to define an equivalence relation $\simeq^\iota_{\mathcal{A}}$ on $U$ that is parameterized by the logical structure $S \in 2\text{−STRUCT}$ to be abstracted.

$$u_1 \simeq^S_{\mathcal{A}} u_2 \;\Leftrightarrow\; \forall p \in \mathcal{A} : S(p)(u_1) = S(p)(u_2)$$

This equivalence relation defines the surjective function $f^S_{\mathcal{A}} : U \to U/\simeq^S_{\mathcal{A}}$ that maps an individual to its equivalence class. We thus have the Galois connection

$$\begin{array}{rcl} \wp(\mathit{State}) = \wp(2\text{−STRUCT}[\mathcal{P}]) & \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} & \wp(3\text{−STRUCT}[\mathcal{P}]) = A \\ \alpha(X) & = & \{f^S_{\mathcal{A}}(S) \mid S \in X\} \\ \gamma(Y) & = & \{S \mid S^\sharp \in Y \wedge S \sqsubseteq^f S^\sharp\} \end{array}$$

where $f^S_{\mathcal{A}}$ is the tight embedding function for logical structures induced by $f^S_{\mathcal{A}} : U \to U/\simeq^S_{\mathcal{A}}$.



(a) a 2-valued structure $S$ that represents a singly-linked list

(b) the canonical abstraction of $S$ with $\mathcal{A} = \{x\}$

Unary predicates associated with variable pointers (e.g., $x$) are depicted with arrows. The other unary predicates $r[n, x]$) are depicted inside nodes for which they evaluate to true. (The meaning of $r[n, x]$ will be explained in Section 5.2.1; see also Table V.) Binary predicate (e.g., $n$) are depicted using arrows linking the two arguments. Solid arrows denote the value 1, dashed arrows denote the value 1/2. Summary nodes (for which $eq = 1/2$) are depicted using double ovals.

Fig. 12. Graphical representation of logical structures that represent memory configurations.

The abstraction function $\alpha$ is referred to as *canonical abstraction*. It defines the *canonical* 3*-valued structures* as those that are the image of canonical abstraction. Figure 12 illustrates the abstraction of a singly-linked list using the predicate $x$ as the unique abstraction predicate. The ordering in $A$ extends the ordering between 3-valued structures as follows: $Y_1 \sqsubseteq Y_2$ iff $\forall S_1^\sharp \in Y_1 : \exists S_2^\sharp \in Y_2 : S_1^\sharp \sqsubseteq S_2^\sharp$.

Thanks to the Embedding Theorem (Theorem 3.2), one can evaluate a logical formula in a 3-valued structure to obtain a conservative result with respect to the structure's concretization as a set of 2-valued structures. Consequently, we can reuse the formulas that specify the concrete operational semantics of statements and conditions (see Section 4): when evaluated in a 3-valued structure, these formulas yield sound approximations (in the abstract lattice $A$) of the concrete transformers.

5.2.1 *Instrumentation Predicates.* As always with abstraction interpretation, there is a danger that as the analysis proceeds, the indefinite value 1/2 will become pervasive. This can destroy the ability to recover interesting information (although soundness is maintained). A key role for improving the precision of the abstraction is played by *instrumentation predicates*, which record auxiliary information in a logical structure. An instrumentation predicate $p$ of arity $k$ is defined by a logical formula $\psi_p(v_1, \ldots, v_k)$ over the core predicate symbols, and captures a property that each $k$-tuple of nodes may or may not possess. Table V lists some instrumentation predicates that are important for the analysis of programs that use type `List`.

If the set of instrumentation predicates is denoted by $\mathcal{I} \subseteq \mathcal{P}$, the concretization function becomes

$$\gamma(S^\sharp) = \left\{ S \in \gamma_{\mathcal{A}}^S(S^\sharp) \middle| \forall p \in \mathcal{I} : [\![ p(v_1, \ldots, v_k) ]\!]_2^S = [\![ \psi_p(v_1, \ldots, v_k) ]\!]_2^S \right\}. \quad (6)$$

The constraint in Eq. (6) that the value of an instrumentation predicate $p$ must match its defining formula $\psi_p$ filters out many concrete structures from consideration, thereby increasing the precision of the abstraction.

Moreover, the use of unary instrumentation predicates as abstraction predicates provides a way to control which concrete individuals are merged together into summary nodes, and thereby to control the amount of information lost by abstraction. For instance, in program-analysis applications, reachability

Table V. Defining Formulas of Instrumentation Predicates Used to Characterize Singly-Linked Lists

| $p$ | Intended Meaning | $\psi_p$ |
|---|---|---|
| $t[n](v_1, v_2)$ | Is $v_2$ reachable from $v_1$ along n fields? | $n^*(v_1, v_2)$ |
| $r[n, q](v)$ | Is $v$ reachable from pointer variable q along n fields? | $\exists\, v_1 : q(v_1) \wedge t[n](v_1, v)$ |
| $c[n](v)$ | Is $v$ on a directed cycle of n fields? | $\exists\, v_1 : n(v, v_1) \wedge t[n](v_1, v)$ |
| $is[n](v)$ | Is $v$ pointed by 2 or more n fields? | $\exists\, v_1, v_2 : \neg eq(v_1, v_2) \wedge n(v_1, v) \wedge n(v_2, v)$ |

Typically, there is a Separate Predicate Symbol $r[n, q]$ for each Pointer Variable q.

properties from specific pointer variables have the effect of keeping disjoint sublists or subtrees summarized separately. This is particularly important when analyzing a program in which two pointers are advanced along disjoint sublists.[6]

When applying the abstract transformer $[\![\mathsf{stm}]\!]$ : 3-STRUCT → 3-STRUCT for statement stm, one could first update the values of the core predicates, and then reevaluate each instrumentation predicate's defining formula in the resulting abstract store. However, this would not provide any additional information. To gain maximum benefit from instrumentation predicates, their value should be computed in some other way. This problem, the *instrumentation-predicate-maintenance problem*, is solved by updating the instrumentation predicates of the post-state as a function of their values in the pre-state. Reps et al. [2003] present an algorithm to generate an appropriate predicate-maintenance formula for each instrumentation predicate $p$, using the (core) predicate-update formulas $\varphi^c_{\mathsf{stm}}$ that define the semantics of stm, together with $p$'s defining formula $\psi_p(v_1, \ldots, v_k)$.

Given the importance of instrumentation predicates that express reachability properties, such as $t[n](v_1, v_2)$ and $r[n, q](v)$ shown in Table V, for maintaining precision under canonical abstraction, there is one limitation of the method from Reps et al. [2003] that is worth mentioning: if $b$ is a core binary predicate, and $t[b]$ is the corresponding reachability predicate, the method from Reps et al. [2003] works best when the modification to $b$ by each concrete transformer is a *unit-size change*, that is, when the transformer changes the value of at most one $b$-tuple. This presents a problem for creating *summary* transformers for procedures, because the net action of a procedure will modify multiple $b$-tuples, in general. Fortunately, the approach to applying procedure summaries developed in this article uses a different approach to maintaining the values of instrumentation predicates than the one presented in Reps et al. [2003] (see Section 6.5).

[6] A method for automatically identifying appropriate instrumentation predicates, using a process of abstraction refinement, is presented in Loginov et al. [2005]. In that paper, the input required to specify a program analysis consists of (i) a program, (ii) a characterization of the inputs, and (iii) a query (i.e., a formula that characterizes the intended output). That work, along with Reps et al. [2003], provides a framework for automating most of the issues related to instrumentation predicates that were explicit obligations of an analysis designer in the original formulation of the 3-valued-logic approach to shape analysis [Sagiv et al. 2002]. See also Loginov [2006].

5.2.2 *Other Operations on Logical Structures.* Several additional operations on logical structures help prevent an analysis from losing precision [Sagiv et al. 2002].

—*Focus* is an operation that can be invoked to elaborate a 3-valued structure, allowing it to be replaced by a set of more precise 3-valued structures (not necessarily images of canonical abstraction) that represent the same set of concrete stores.

—*Coerce* is a clean-up operation that may "sharpen" a 3-valued structure by setting an indefinite value (1/2) to a definite value (0 or 1), or discard a structure entirely if the structure exhibits some fundamental inconsistency (e.g., it cannot represent any possible concrete store).

Because the Embedding Theorem applies to any pair of structures for which one can be embedded into the other, it is not necessary to perform canonical abstraction after the application of each abstract transformer. To ensure that abstract interpretation terminates, it is only necessary that canonical abstraction be applied as a widening operator somewhere in each loop, for example, at the target of each backedge in the CFG.

## 6. REPRESENTING AND ABSTRACTING RELATIONS BETWEEN MEMORY CONFIGURATIONS

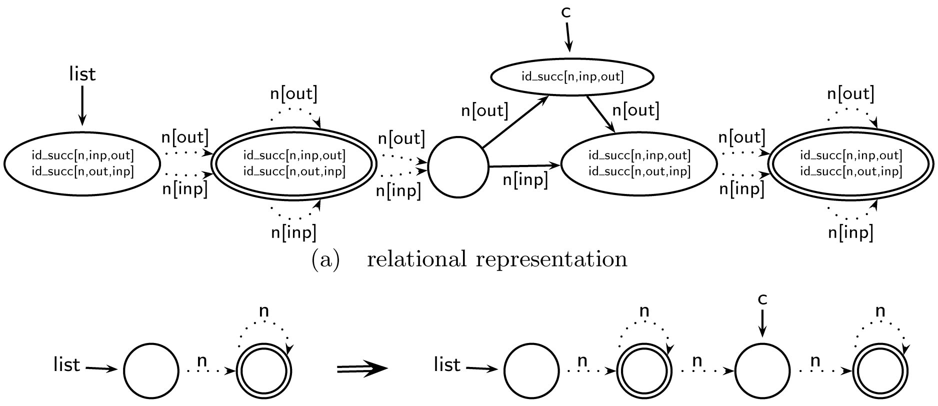
### 6.1 Motivation

As discussed more thoroughly in Section 7 and Section 9, there are two main approaches to interprocedural static analysis: the *functional* and *operational* approaches [Sharir and Pnueli 1981]. In this article, we follow the functional approach (also known as the *relational* approach). A key aspect of the functional approach is that it computes *procedure summaries*. It computes a predicate transformer for each node of the program by finding the smallest fixpoint of a set of equations over predicate transformers. During this process, the effect of a call to procedure $P$ at a call site $c$ is handled by composing the predicate transformer for $c$ with the predicate transformer for $P$. (The predicate transformer for $P$ is the predicate transformer for the exit node of $P$.) When the fixpoint solution is obtained, the predicate transformer for $P$ is the procedure summary for $P$. In this article such predicate transformers will be viewed as relations.

The main point here is that the ability to represent and abstract *relations* between memory configurations is fundamental for capturing the input/output behavior of a procedure. This section shows how representations for relations between memory configurations that are represented as logical structures can be created. This representation is the basis of the interprocedural shape analysis described in the next section.

### 6.2 Principles of the Representation

We now return to the discussion from Section 2 about two ways to represent and abstract relations between concrete program states, when a program state



(a) relational representation

(b) tabulated representation

Fig. 13. Two abstractions of the relation between an input list and an output list in which a new cell pointed to by c has been inserted (using destructive updating) somewhere in the middle of the list. Predicates $n[inp]$ and $n[out]$ represent the valuations of the $n$ predicate before and after the insertion, respectively.

is a 2-valued structure. The first approach described in Section 2 involved representing relations between concrete program states as sets of pairs of 2-valued structures.

This point of view leads to a simple abstraction, where abstract relations are (sets of) pairs of 3-valued structures obtained by canonical abstraction; see Figure 13(b). However, this solution is unsatisfactory for the following reasons.

- There is a technical difficulty: as explained in Remark 4.1, logical structures are implicitly defined up to a permutation of individuals. As explained in Section 2, this leads to a loss of information compared with first pairing and then abstracting.[7]

  With this representation it is also difficult to implement the application of a predicate transformer (sets of pairs) to an input predicate (a set of logical structures).
- From an efficiency point of view, applying such a solution to a complex abstract domain like 3-valued structures would often lead to combinatorial explosion.[8]

Fortunately, another approach is possible. We will proceed by analogy with an approach used when abstracting sets of vectors $X \subseteq \mathbb{R}^n$ and sets of relations

[7] In concrete structures, identity of individuals is preserved in any given run of a procedure. The problem with abstraction-and-pairing is that the identity of the abstract individual to which a given concrete individual is mapped is not necessarily the same when different concrete structures are abstracted. The canonical name for $u$ in $S_1^\sharp$ on entry to a procedure has no a priori fixed relationship to the canonical name in a structure $S_2^\sharp$ that arises at the exit of the procedure.

[8] Even with intraprocedural analysis using single structures, combinatorial explosion needs to be carefully controlled by choosing a suitable set of abstraction predicates.

$R \subseteq \mathbb{R}^n \times \mathbb{R}^n$ between such vectors. Sets of vectors can be abstracted with convex polyhedra [Cousot and Halbwachs 1978].

$$\wp(\mathbb{R}^n) \xleftarrow{\gamma} \mathsf{Pol}[n]$$

It is well known that a good approach to abstracting relations between vectors is not to consider pairs of polyhedra, but to view relations between $n$-dimensional vectors as sets of $2n$-dimensional vectors, and to consider polyhedra in $2n$ dimensions.

$$\wp(\mathbb{R}^n \times \mathbb{R}^n) \xleftarrow{\gamma} \mathsf{Pol}[2n]$$

Indeed, a relation like $\vec{x} = \vec{x}'$ cannot be finitely represented with pairs of polyhedra, but is very easily represented with a $2n$-dimensional polyhedron. Composing two such relations $P_1, P_2 \in \mathsf{Pol}[2n]$ is also easy: one computes the intersection

$$P_{12}(\vec{x}, \vec{x}', \vec{x}'') = P_1(\vec{x}, \vec{x}', -) \cap P_2(-, \vec{x}', \vec{x}'') \in \mathsf{Pol}[3n],$$

and then projects out the $\vec{x}'$ variables in $P_{12}$.

Coming back to 2-valued logical structures $(U, \iota : \mathcal{P} \to \bigcup_k(U^k \to \mathbb{B}))$, an analogy can be drawn with polyhedra by considering each predicate symbol in a logical structure over a vocabulary $\mathcal{P}$, where $|\mathcal{P}| = n$, to correspond to a dimension in an $n$-dimensional vector. Thus, we will use logical structures over the duplicated vocabulary $\mathcal{P} \uplus \mathcal{P}'$ to represent relations between logical structures over vocabulary $\mathcal{P}$. Observe that the representation of concrete and abstract relations is unified by the notion of 3-valued structures, as before.

Taking the analogy further, the existential quantification of a dimension in a set of vectors $X \subseteq \mathbb{R}^n$ corresponds to assigning the value $\mathbf{1/2}$ to all tuples of a predicate. With the addition of a meet operation on 3-valued structures (described in Section 6.5.2), we will be able to implement relation composition on two-vocabulary structures, in a manner similar to convex polyhedra in $2n$ dimensions.

*Example* 6.1. Figures 13(a) and 13(b) illustrate the relational and tabulated representations, respectively, of a relation between input lists pointed to by a pointer variable `list` and output lists obtained by the insertion of a cell pointed to by pointer `c`.[9]

The meanings of the *relational instrumentation predicates* displayed inside the nodes in Figure 13(a) are explained in Section 6.4. They allow the analysis to track whether the fields of some cells have been modified or not.

Observe that the relational representation provides more information, because each cell is tracked individually in the representation. For instance, in Figure 13(b), the information that the output list contains exactly one more cell than the input list is lost. Furthermore, with the tabulated representation,

[9] To reduce clutter, we have omitted certain information from Figure 13(a); in particular, values have been omitted for some of the standard list predicates given in Table V, and therefore the reason why certain non-summary nodes have been kept separate from the summary nodes may not be apparent. This is just to simplify the diagram; the actual system has additional information not shown in Figure 13(a) that controls which collections of nodes are summarized.

there is no way to determine whether the cells in the output list have been permuted from their order in the input list. In contrast, with the relational representation and the use of the relational instrumentation predicates, it is possible to record the fact that the fields of some cells have not been mutated.

### 6.3 Structure of the Vocabulary

In this section, we define the vocabularies that are used when two-vocabulary logical structures are used to represent relations between logical structures.

Because our analysis method will use relation composition (see Eq. (12) in Section 7), we actually need three vocabularies. For each original predicate $p \in \mathcal{P}$, we will define three predicates $p[inp]$, $p[out]$, and $p[tmp]$. A logical structure that represents a relation will use only $p[inp]$ and $p[out]$ predicates. The $p[tmp]$ predicates (which will be used for computing compositions as explained shortly) are irrelevant outside of composition. The "irrelevancy" of a predicate corresponds to "undefinedness", and will be modeled in a 3-valued structure using the value **1/2**. We will refer to the labels *inp*, *out*, and *tmp* as *modes*.

We have already distinguished, among predicates, core predicates from instrumentation predicates: $\mathcal{P} = \mathcal{C} \cup \mathcal{I}$. Moreover, among core predicates, we have distinguished predicates related to the local state and those related to the global state: $\mathcal{C} = \mathcal{L} \cup \mathcal{G}$. The vocabulary of core predicates will now contain:

—three sets of predicates corresponding to global core predicates in $\mathcal{G}$: $\mathcal{G}[inp]$, $\mathcal{G}[out]$, and $\mathcal{G}[tmp]$;

—the set of local core predicates $\mathcal{L}$.

We will assume that the formal input parameter of a procedure is not modified in the procedure, so as to obtain at the exit node of the procedure a relationship between the values of predicates in $\mathcal{G}[inp] \cup \{\mathsf{fpi}\}$ and predicates in $\mathcal{G}[out] \cup \{\mathsf{fpo}\}$. The other local variables may be forgotten at the exit node.

The case of an instrumentation predicate $p$ is a bit more complex, because it depends on the predicates involved in its defining formula $\psi_p$. If $\psi_p$ involves at least one global predicate, the vocabulary will include three copies of the instrumentation predicate $p$: $p[inp]$, $p[out]$, and $p[tmp]$. For instance, the vocabulary will include three copies of the reachability predicate $r[n, q](v)$ defined in Table V, because we need to characterize a cell by its reachability properties from the pointer variable $q$ through $n$ links both at the entry of the procedure and at the current control point.

We can now give the precise definition of 3-valued structures $S^\sharp = (U^\sharp, \iota^\sharp) \in$ 3$-$STRUCT$[\mathcal{P}[inp] \cup \mathcal{P}[out]]$ in terms of a relation $R \subseteq \big(\text{2-STRUCT}[\mathcal{C}]\big)^2$:

$$\gamma_r(S^\sharp) = \left\{ ((U, \iota_1), (U, \iota_2)) \;\middle|\; \begin{array}{l} \exists S = (U, \iota) \in \gamma(S^\sharp) : \\ \left\{ \begin{array}{l} \forall p \in \mathcal{G}[inp] : \iota_1(p) = \iota(p[inp]) \\ \forall p \in \mathcal{G}[out] : \iota_2(p) = \iota(p[out]) \\ \forall p \in \mathcal{L} : \iota_1(p) = \iota_2(p) = \iota(p) \end{array} \right. \end{array} \right\}$$

where the concretization function $\gamma$ is defined by Eq. (6).

## 6.4 Relational Instrumentation Predicates

To prevent loss of essential information, we also need specific instrumentation predicates to capture properties that relate $p[inp]$ predicates and $p[out]$ predicates. We call such multi-vocabulary instrumentation predicates *relational instrumentation predicates*.

In particular, it will be essential to capture accurately the identity relationship (see Section 7.1, Eq. (11)). As a consequence, we always use the unary predicates $id\_succ[n, m_1, m_2]$ and $id\_pred[n, m_1, m_2]$, where $m_1, m_2 \in \{inp, out\}$ and $m_1 \neq m_2$, to record information about the values of different modes of predicate $n$, such as whether the value of predicate $n[m_1]$ implies $n[m_2]$. These are defined by

$$\begin{aligned} id\_succ[n, m_1, m_2](v) &= \forall v_1 : (n[m_1](v, v_1) \Rightarrow n[m_2](v, v_1)) \\ id\_pred[n, m_1, m_2](v) &= \forall v_1 : (n[m_1](v_1, v) \Rightarrow n[m_2](v_1, v)). \end{aligned}$$

*Example* 6.2. In Figure 13(a), the fact that $id\_succ[n, inp, out](v)$ and $id\_succ[n, out, inp](v)$ both hold for the two summary nodes captures the fact that the concrete memory cells represented by these summary nodes have not been reordered. More generally, the value of $id\_succ[n, m_1, m_2]$ on the different nodes allows to capture precisely that the only transformation performed on the list is the addition of the new cell. (Looking ahead to Figure 15(a), the fact that $id\_succ[n, inp, tmp](v)$ and $id\_succ[n, tmp, inp](v)$ hold globally captures the condition that the $n[inp]$ and $n[tmp]$ predicates are identical.)

Generally speaking, relational instrumentation predicates are essential to preserving relational information that would otherwise be lost when concrete nodes are merged into summary nodes.

Some additional constraint rules related to these relational instrumentation predicates are also needed for the relation composition operation defined in Section 6.5. These constraint rules express logical consequences between relational instrumentation predicates. For instance, the rule

$$id\_succ[n, m_1, m_2](v) \wedge id\_succ[n, m_2, m_3](v) \Rightarrow id\_succ[n, m_1, m_3](v)$$

for $m_1 \neq m_2 \neq m_3$ is standard for capturing the fact that the composition of two identity relations is the identity relation. At present time such rules are provided manually.

Depending on the procedures in the analyzed program and their semantics, one may need additional relational instrumentation predicates and constraint rules. For the list-reversal example of Figure 1, Section 8.1 discusses the relational instrumentation predicates used to capture the fact that the list has been reversed.

## 6.5 Relation Composition

As mentioned in Section 6.2, relation composition can be defined in terms of meet and projection operations. In the notation from Section 2, the composition

$S^{\langle ',''\rangle} \circ S^{\langle \cdot,'\rangle}$ of the transformations represented by two two-vocabulary structures $S^{\langle \cdot,'\rangle}$ and $S^{\langle ',''\rangle}$ is performed as follows.

$$\begin{aligned} S^{\langle ',''\rangle} \circ S^{\langle \cdot,'\rangle} &= \text{ project}_{1,3}\big(S^{\langle 1/2,',''\rangle} \sqcap S^{\langle \cdot,',1/2''\rangle}\big) \\ &= S^{\langle \cdot,''\rangle} \end{aligned} \tag{7}$$

We define the projection and meet operations next, and discuss their interaction with instrumentation predicates.

6.5.1 *The Projection Operation.* The existential quantification of a (core) predicate symbol $p_0$ in a 2-valued logical structure $S = (U, \tau)$ is formally defined as the disjunction of all the possible values $\{\mathbf{1}, \mathbf{0}\}$ for all tuples of the predicate $p_0$ in $S$, leading to a set of 2-valued structures.

$$\exists p_0 : S = \{S' = (U, \tau') \mid \forall p \in \mathcal{P} \setminus \{p_0\} : \tau'(p) = \tau(p)\}$$

Now consider existential quantification in a 3-valued logical structure $S^\sharp$. The goal is to create a 3-valued structure that overapproximates the result of existential quantification in all 2-valued structures that $S^\sharp$ represents.

When $S^\sharp$ contains no instrumentation predicates, existential quantification can be modeled *exactly* by assigning the value $\mathbf{1/2}$ to all tuples of the predicate $p_0$, as follows:

$$(\exists p_0 : S) = (U, \tau'),$$

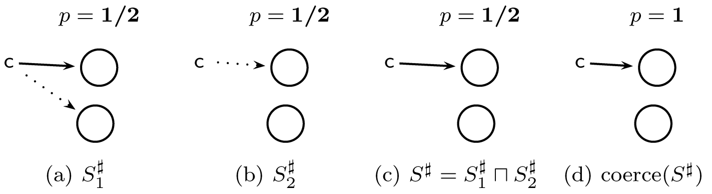
where $\tau'$ is defined by $\forall \vec{u} \in U^* : \tau'(p_0)(\vec{u}) = \mathbf{1/2} \;\wedge\; \forall p \in \mathcal{P} \setminus \{p_0\} : \tau'(p) = \tau(p)$. This operation can be implemented with a predicate-update formula (Section 5.2). Applying the concretization operation $\gamma : \wp(\text{ 3-STRUCT}) \rightarrow \wp(\text{ 2-STRUCT})$ gives back the disjunction of 2-valued structures.

Matters are slightly different when we consider a 3-valued logical structure equipped with instrumentation predicates. Consider $S^\sharp \in$ 3-STRUCT[$\mathcal{P}$], where $\mathcal{P} = \mathcal{C} \cup \mathcal{I}$ has core predicates $\mathcal{C}$ and instrumentation predicates $\mathcal{I}$. Quantifying out a core predicate $c$ alone may not be sufficient to drop all information about $c$: in particular, every instrumentation predicate whose defining formula involves $c$ provides (a degree of) redundant information about $c$; hence, all instrumentation predicates whose defining formula involves $c$ should also be quantified out.[10]

Projecting a logical structure in 3-STRUCT[$\mathcal{P}[inp] \cup \mathcal{P}[out] \cup \mathcal{P}[tmp]$] onto the subspace 3-STRUCT[$\mathcal{P}[inp] \cup \mathcal{P}[out]$] is thus equivalent to the existential quantification of all $p[tmp]$ predicates, for $p \in \mathcal{P}$, as well as all relational instrumentation predicates that involve a predicate in $p[tmp]$.

This operation on 3-valued structures is extended in the standard way to our abstract domain $\wp($ 3-STRUCT[$\mathcal{P}[inp] \cup \mathcal{P}[out] \cup \mathcal{P}[tmp]$]) that manipulates sets of such structures.

[10] Quantifying out $c$ and all instrumentation predicates whose defining formula involves $c$ might not be the *best* correct approximation of quantifying out $c$ in all concrete structures represented by $S^\sharp$ if the defining formula $\psi_p$ of an instrumentation predicate $p$ has a *syntactic* dependence on $c$ without involving a true semantic dependence, for instance, if we have $\psi(p)(\vec{v}) = \ldots \wedge (c(\vec{v}) \vee \neg c(\vec{v}))$.



(a) $S_1^\sharp$ (b) $S_2^\sharp$ (c) $S^\sharp = S_1^\sharp \sqcap S_2^\sharp$ (d) coerce($S^\sharp$)

Fig. 14. Applying the Coerce operation after the meet operation. $c$ is a core predicate, $p$ a nullary instrumentation predicate defined by $p = \exists v : (c(v) \wedge (\forall v' : v \neq v' \Rightarrow \neg c(v')))$.

6.5.2 *The Meet Operation.* The meet operation is first defined as the greatest-lower-bound operation induced by the approximation order in the lattice 3-STRUCT[$\mathcal{P}$]. It is then extended to the abstract domain $\wp$( 3-STRUCT[$\mathcal{P}$]). Arnold et al. [2006] shows that in general the first operation is NP-complete. However, Arnold et al. [2006] provide an algorithm based on graph matching that performs rather well in practice. This is discussed in more detail in Section 8.3.

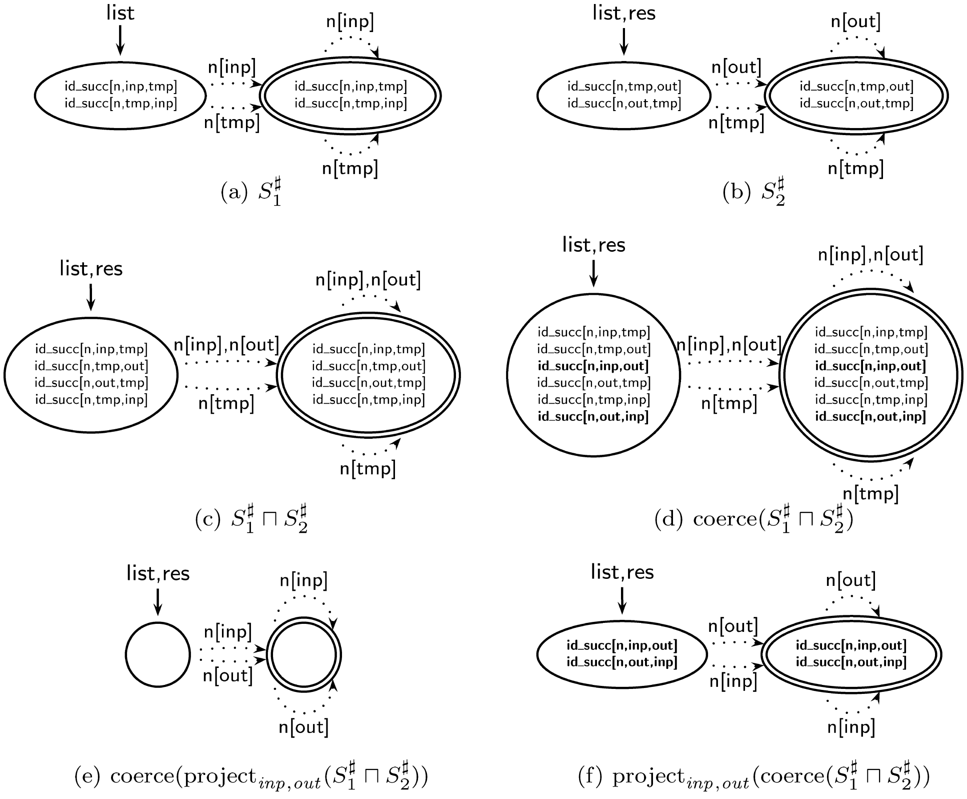
The effect of the meet operation on instrumentation predicates deserves a further remark: In the context of *abstract* structures, it should be combined with the Coerce operation discussed in Section 5.2.2, which propagates logical consequences between (core and instrumentation) predicates. Indeed, the standard meet operation performs a logical meet without exploiting the defining formulas of instrumentation predicates: instrumentation predicates are just treated as independent core predicates.

Consider the example of Figure 14. It returns the structure depicted in Figure 14(c), where $p$ holds the indefinite value 1/2. However, performing a semantic reduction on $S^\sharp$ using Coerce leads to $p$ obtaining the definite value 1, as shown in Figure 14(d). In this case, Coerce used constraint rules derived from the defining formula of $p$ to infer that $p$ must have the value 1. (See Sagiv et al. [2002, Section 6.4] for more details about the use of constraint propagation during Coerce.)

This aspect is even more important in the context of multivocabulary logical structures that are combined for relation composition; see Figure 15. As discussed in Section 6.4, the multivocabulary logical structures that we work with are typically equipped with relational instrumentation predicates and related constraint rules. To retain precision, it is necessary to make sure that logical consequences of the predicates in the vocabulary to be dropped have been incorporated into the predicates of the other vocabularies *before* projection. Figure 15 illustrates this point when the id_succ[n,m1,m2] relational instrumentation predicates and their related constraint rules as defined in Section 6.4 are active. It shows that applying the Coerce operation before projection is the key to obtaining the fact that the resulting relation in Figure 15(f) is the identity relation.

As a consequence, the abstract meet operation between 3-valued structures is defined as

$$S_1^\sharp \sqcap^\sharp S_2^\sharp = \text{coerce}(S_1^\sharp \sqcap S_2^\sharp),$$



(a) $S_1^\sharp$

(b) $S_2^\sharp$

(c) $S_1^\sharp \sqcap S_2^\sharp$

(d) coerce$(S_1^\sharp \sqcap S_2^\sharp)$

(e) coerce(project$_{inp,out}(S_1^\sharp \sqcap S_2^\sharp)$)

(f) project$_{inp,out}$(coerce$(S_1^\sharp \sqcap S_2^\sharp)$)

Fig. 15. Applying the Coerce operation in relation composition.

where $\sqcap$ is the standard meet on 3-valued structures, and the abstract relation-composition operation (Eq. (7)) is redefined as

$$S^{\langle ',''\rangle} \circ S^{\langle \cdot,'\rangle} = \text{project}_{1,3}(S^{\langle 1/2,',''\rangle} \sqcap^\sharp S^{\langle \cdot,',1/2''\rangle}) \\ = S^{\langle \cdot,''\rangle}. \tag{8}$$

The use of the abstract meet operation in Eq. (8) addresses a problem that was mentioned in Section 5.2.1: the instrumentation-predicate-maintenance formulas created by finite differencing [Reps et al. 2003] are able to maintain definite values for instrumentation predicates that express reachability properties only for unit-size changes to core predicates. However, procedure summaries can involve non-unit-size changes to core predicates. We side-step this problem by using abstract meet, rather than a method that involves finite differencing, to implement abstract relation composition.

## 7. INTERPROCEDURAL SHAPE ANALYSIS

Our interprocedural shape analysis is based on a variant of the functional approach to interprocedural analysis [Cousot and Cousot 1977; Sharir and Pnueli 1981; Knoop and Steffen 1992], in which the two computation steps referred to in Section 6.1 are merged into a single step. Jeannet and Serwe [2004] show

how the functional approach can be derived as an abstract interpretation of the standard operational semantics, modeled using a stack of activation records. Once the interprocedural semantics is defined in this way, a second abstraction step may be used to abstract the data (in our case, the values of variables and linked memory cells).

In this section, we start directly from the derived forward relational semantics obtained by abstract interpretation of the standard operational semantics, as described in Jeannet and Serwe [2004]. In Section 7.1, we first instantiate this forward relational semantics for the case where relations between memory configurations are represented as sets of pairs. In Section 7.2, we reformulate it for the case where relations are represented and abstracted with the two-vocabulary structures defined in Section 6, so as to obtain the effective dataflow equations used by our analysis. Finally, in Section 7.3, we discuss how these dataflow equations can be modified so that their solutions can be obtained more rapidly. (Experimental results with the latter technique are presented in Section 8.4.)

### 7.1 Forward Relational Semantics

In the forward relational semantics, each node of the program's CFG is associated with a relation between

—the states reachable at the entry node of the current procedure, and

—the states reachable at the current node of the procedure.

The relational semantics is defined as the least fixpoint of a system of equations over such relations.

Each procedure is viewed as a pure function taking inputs and returning outputs, without performing any side effect on the global store. However, the programs that we consider *do* modify the global store, defined by the heap and the value of global variables. To account for this, at the semantic level we include the heap and the global variables as implicit input and output parameters of the functions, in addition to the explicit input and output parameters.

*Notation.* For the time being, we represent relations between concrete memory configurations as sets of pairs of 2-valued structures. Thus, we define the function $R : N \to \wp(\mathit{State} \times \mathit{State})$, which maps each node of the CFGto a relation over states.

States are represented as 2-valued logical structures over core predicates. Among the core predicates, some predicates represent information about the local state of a procedure (i.e., the values of local variables), while other predicates represent information about the global state of the program, that is, the structure of the heap and the values of global variables. We thus decompose the set of core predicates into *local* and *global* predicates.

$$\mathcal{C} = \mathcal{G} \cup \mathcal{L}$$

*Intraprocedural operations.* An edge $n \rightarrow n'$ of the CFG labeled with a statement $\mathsf{stm}$ or a condition $\mathsf{cond}$ generates the following equations, respectively.

$$R(n') \supseteq \{(S, S'') \mid (S, S') \in R(n) \land S'' = [\![\mathsf{stm}]\!](S')\} \tag{9}$$

$$R(n') \supseteq \{(S, S'') \mid (S, S') \in R(n) \land S'' \in [\![\mathsf{cond}]\!](\{S'\})\} \tag{10}$$

Intuitively, the current relation is composed with the relation induced by the semantics of the operation. We use inclusions in the equations because several edges may have $n'$ as their target.

*Procedure calls.* In a procedure call, modeled by a call-to-start edge $(c, s)$ labeled by an expression $\langle \mathsf{call\ apo} = P_i(\mathsf{api})\rangle$, the current global state and the actual parameter are passed to the callee, while the other local variables become undefined. One generates the identity relation from the obtained reachable set of states.

$$R(s) = \left\{ (T, T) \,\middle|\, (S, S') \in R(c) \land \left| \begin{array}{rrcl} & T(\mathsf{fpi}) & = & S'(\mathsf{api}) \land \\ \forall p \in \mathcal{G} : & T(p) & = & S'(p) \end{array} \right| \right\} \tag{11}$$

Note that an undefined predicate is modeled as: "any value is possible".

*Procedure returns.* This is the most complex operation. We assume that an exit-to-return edge $(e, r)$ is labeled by $\langle \mathsf{ret\ apo} = P_i(\mathsf{api})\rangle$, and that $(e, r)$'s corresponding call-to-start edge is $(c, s)$ (i.e., $call(r) = c$). The processing of a procedure return consists of the following steps:

— composing the relation $R(c)$ at the corresponding call node $c$ with the relation $R(e)$ at the exit node of the callee, to create the global state at $r$;
— taking the local state at the call node and modifying it with the assignment of the actual output parameter at the exit node, to create the local state at $r$.

$$R(r) = \left\{ (S, W) \,\middle|\, \begin{array}{lrl} & & (S, S') \in R(c) \ \land \ (T, T') \in R(e) \\ \land & \forall p \in \mathcal{G} : & S'(p) = T(p) \ \land \ S'(\mathsf{api}) = T(\mathsf{fpi}) \\ \land & \forall p \in \mathcal{L} \setminus \{\mathsf{apo}\} : & W(p) = S'(p) \\ \land & \forall p \in \mathcal{G} : & W(p) = T'(p) \ \land \ W(\mathsf{apo}) = T'(\mathsf{fpo}) \end{array} \right\} \tag{12}$$

In the preceding equations, for $(S, S')$ and $(T, T')$ to be composable, the states $S'$ and $T$ must agree on the input parameters (actual and formal) and the global state. In the new state $W$, the values of local variables except the actual output parameter are inherited from $S'$, while the global state and the value of the actual output parameter are taken from $T'$.

*The initial set of relations.* Normally, the analysis starts in an initial state (here, a relation). Assuming that the set of possible memory configurations at the start node of the main procedure is $X$, we add the inclusion

$$R(s_{\mathsf{main}}) \supseteq \{(S, S) \mid S \in X\}. \tag{13}$$

*Reachable states.* The set that we want to compute is the least fixpoint of Eqs. (9), (10), (11), (12), and (13). This defines a framework for interprocedural dataflow analysis.

—A given analysis is obtained by instantiating these equations for a suitable abstract domain.

—At each control-flow graph node, the fixpoint solution captures the relation between the reachable states at the entry of the current procedure and the reachable states at the current node.

—The states reachable at each node $n$ can thus be extracted by projecting the relation $R(n)$ onto its second component.

Eqs. (9), (10), (11), (12), and (13) are a particular version of the equations given in Jeannet and Serwe [2004], except that the global state is passed back and forth explicitly. (Also, here we merge the two sets of activation records that were kept separate in Jeannet and Serwe [2004] to support backward analysis.) The soundness of the semantics with respect to the standard operational semantics is proven in Jeannet and Serwe [2004] by using abstract interpretation.

### 7.2 Dataflow Equations

In Section 6, we showed how to represent relations between logical structures more efficiently and to abstract them more precisely with two-vocabulary structures. We thus instantiate the equations of Section 7.1 with this better representation.

*Intraprocedural operations.* Eqs. (9) and (10) are replaced by

$$\begin{aligned} R(n') &\supseteq [\![\mathsf{stm}]\!](R(n)) \\ R(n') &\supseteq [\![\mathsf{cond}]\!](R(n)) \end{aligned}$$

except that predicate-update formulas and precondition formulas in functions $[\![\mathsf{stm}]\!]$ and $[\![\mathsf{cond}]\!]$ defined by Eqs. (4) and (5) are modified by replacing global predicates $p \in \mathcal{G}$ with predicates $p[out] \in \mathcal{G}[out]$. For instance, in the case of the statement `x->n := NULL`, the predicate-update formula becomes.

$$n'[out](v_1, v_2) = n[out](v_1, v_2) \wedge \neg x(v_1).$$

*Procedure calls.* Eq. (11) is replaced by

$$R(s) = \left\{ T \;\middle|\; S \in R(c) \wedge \left( \begin{array}{lrcl} & T(\mathsf{fpi}) & = & S(\mathsf{api}) \\ \wedge\ \forall p \in \mathcal{L} \setminus \{\mathsf{fpi}\} : & T(p) & = & \mathbf{1/2} \\ \wedge \qquad \forall p \in \mathcal{G} : & T(p[inp]) = T(p[out]) & = & S(p[out]) \end{array} \right) \right\}.$$

*Procedure returns.* We proceed in three steps to implement Eq. (12). First we take the relation $R(e)$ at the exit node of the callee and transform it by eliminating local variables that are not formal input or output parameters, and by setting the values of $p[tmp]$ predicates to the values of $p[inp]$ predicates.

$$R'(e) = \left\{ S' \;\middle|\; \exists S \in R(e) : \left\{ \begin{array}{rrcl} \forall p \in \mathcal{L} \setminus \{\mathsf{fpi}, \mathsf{fpo}\} : & S'(p) & = & \mathbf{1/2} \\ \forall p \in \mathcal{G} : & S'(p[inp]) & = & \mathbf{1/2} \\ \forall p \in \mathcal{G} : & S'(p[tmp]) & = & S(p[inp]) \end{array} \right. \right\}$$

We also take the relation $R(c)$ at the call node, set the values of $p[tmp]$ predicates to the values of $p[out]$ predicates, and equate formal and actual input parameters. (To simplify the presentation, we assume that there are no name conflicts.)

$$R'(c) = \left\{ S' \;\middle|\; S \in R(c) \wedge \left( \begin{array}{rcl} S'(\mathsf{fpi}) & = & S(\mathsf{api}) \\ \wedge\ \forall p \in \mathcal{G}:\ S'(p[tmp]) & = & S(p[out]) \\ \wedge\ \forall p \in \mathcal{G}:\ \ S'(p[out]) & = & 1/2 \end{array} \right) \right\}$$

The last step consists of combining $R'(c)$ and $R'(e)$ by taking their meet, assigning the formal output parameter to the actual parameter, and then forgetting $p[tmp]$ predicates and the formal output parameter of the callee.

$$R(r) = \left\{ S' \;\middle|\; S \in R'(c) \sqcap^{\sharp} R'(e) \wedge \left( \begin{array}{rcl} S'(\mathsf{apo}) & = & S(\mathsf{fpo}) \\ S'(\mathsf{fpi}) & = & \mathbf{1/2} \\ \wedge\ \forall p \in \mathcal{G}:\ S'(p[tmp]) & = & \mathbf{1/2} \end{array} \right) \right\} \quad (14)$$

The meet forces the relations $R'(c) \in$ 3-STRUCT$[\mathcal{P}[inp] \cup \mathcal{P}[tmp]]$ and $R'(e) \in$ 3-STRUCT$[\mathcal{P}[tmp]\cup\mathcal{P}[out]]$ to agree on global $p[tmp]$ predicates, and on actual and formal parameters.

With the exception of the meet operation, all operations can be implemented using predicate-update formulas (refer to Section 4.3). We do not specify in the preceding equations how instrumentation predicates are updated; the implementation mainly uses the automatically generated predicate-maintenance formulas created by finite differencing [Reps et al. 2003], although for some simple cases and for instrumentation predicates that involve only one mode, they were provided manually. In particular, for the procedure-call operations, we provided manually the values of relational instrumentation predicates that model the identity relationship.

### 7.3 Impact of the Form of the Dataflow Equations on Precision and Efficiency

We now compare our one-phase approach to interprocedural analysis to a two-phase approach that is in the spirit of Sharir and Pnueli [1981] and Knoop and Steffen [1992]. At the concrete level, the two approaches are semantically equivalent; however, at the abstract level the one-phase approach can yield more precise answers because the abstract operations are no longer exact. We exploited this difference to develop an optimization that, in practice, speeds up the convergence of our one-phase analysis while still retaining its precision advantages (see the experimental results presented in Section 8.4).

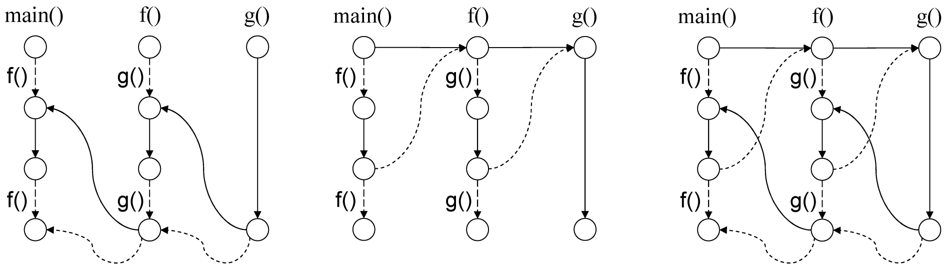
Our forward relational semantics can be sketched as follows.

| | |
|---|---|
| $R(n')\supseteq F_{\mathsf{instr}}(R(n))$ | Intraprocedural statement/condition |
| $R(s_{main})\supseteq \mathrm{Id}_{State}$ | Uninitialized state at start |
| $R(s)\supseteq \mathrm{Id}_{codom(R(c))}$ | Procedure call, with call-to-start edge $(c, s)$ |
| $R(r)\supseteq F_{\mathsf{Combine}}(R(c), R(e))$ | Procedure return, with call-site $c$ and exit-to-return edge $(e, r)$ |

where $\mathrm{Id}_X$ denotes the identity relation restricted to the domain $X$ and $codom(R)$ denotes the projection of a relation $R$ on its codomain. In contrast, the two-phase method involves solving two equation systems in succession.

$$R(n') \supseteq F_{\mathsf{instr}}(R(n))$$
$$R(s_{main}) \supseteq \mathrm{Id}_{State}$$

| | | |
|---|---|---|
| $R(s) \supseteq \mathrm{Id}_{State}$<br>$R(r) \supseteq F_{\mathsf{Combine}}(R(c), R(e))$ | $R(s) \supseteq \mathrm{Id}_{codom(R(c))}$<br>$R(r) \supseteq F_{\mathsf{Combine}}(R(c), \mathcal{R}(e))$ | $R(s) \supseteq \mathrm{Id}_{codom(R(c))}$<br>$R(r) \supseteq F_{\mathsf{Combine}}(R(c), R(e))$ |



(a) phase I, bottom-up (b) phase II, top-down (c) combination with forward relational semantics

Fig. 16. Inequation systems and induced dependences between variables. Solid and dashed lines are used to distinguish between the first and second calls to `f()` and `g()`.

The first system, which defines the so-called bottom-up phase, computes procedure summaries that are valid for any input, instead of being specialized to the reachable inputs of the callee. The second system, which defines the so-called top-down phase, computes reachability information using the procedure summaries $\mathcal{R}(e)$ computed by the first phase. The corresponding equations are given in Figure 16.

The advantage of combining the two phases into a single phase (as done in Section 7.1) is that the one-phase approach can yield more precise answers (because it computes procedure summaries that are specialized to the reachable inputs of the callee).

The one-phase approach may converge more slowly than the two-phase approach because the one-phase equation system is more intricate. However, it is possible to speed up the convergence of the one-phase analysis while retaining its precision advantage as follows: it is sound to replace the inequation $R(s) \supseteq \mathrm{Id}_{codom(R(c))}$ associated with call-to-start edges by $R(s) \supseteq \mathrm{Id}_{codom(R(c)) \cup X^0(c)}$ for any $X^0(c)$. The idea is to choose $X^0(c)$ to be a set of states that is very likely to be reachable (e.g., for a list-manipulating procedure, the set of well-formed lists). Because it may take several iteration steps for the solver to obtain this information and to propagate it further, adding it right from the beginning may speed up convergence. From a semantic point of view, this is equivalent to starting the iterative fixpoint computation from a higher initial value for $R(s)$, which becomes $\mathrm{Id}_{X^0(c)}$ instead of $\bot$. Two cases can occur.

—$X^0(c)$ contains only reachable inputs of the callee, and the initial value of $R(s)$ is still smaller than the smallest solution of the original equation system; in this case, there is no impact on precision, and we gain a convergence speedup.

—$X^0(c)$ contains some unreachable inputs of the callee, which can have a negative impact on precision (with respect to the precision of the one-phase approach). However, in the limit (i.e., when $X^0(c) = State$), one obtains the equations of phase I of the two-phase approach: the propagation from call nodes to start nodes of only reachable abstract values is completely eliminated. In this case, the precision of the one-phase approach degenerates to that of phase I of the two-phase approach (see Figure 16).[11]

The results of our experiments with this optimization are presented in Section 8.4.

## 8. IMPLEMENTATION AND EXPERIMENTS

To perform interprocedural shape analysis by the method that is described in Section 7, we created a modified version of TVLA [Lev-Ami and Sagiv 2000], an existing shape-analysis system, to allow it to support the following features.

— We replaced the built-in notion of an intraprocedural CFG by the more general notion of *equation system*, in which transfer functions may depend on more than one variable. This modification was needed for implementing the return operation (Eq. (14)).

— We also designed a more general language in which to specify equation systems.

These modifications, originally performed in 2003 [Jeannet et al. 2004], were made to the version of the TVLA system as it existed in 2003 [Lev-Ami and Sagiv 2000]. Later, we extended the modified system to incorporate the algorithm for the meet operator described in Arnold et al. [2006]. It is this version of TVLA that we used in the experiments reported here.

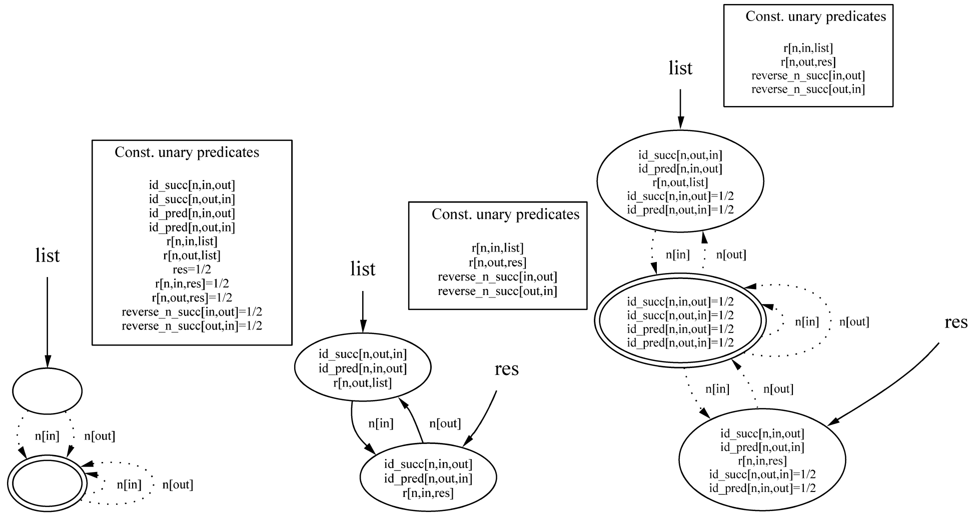
This section is organized as follows: Section 8.1 discusses the analysis of the recursive list-reversal procedure from Figure 1; Section 8.2 describes our experiments on a variety of list manipulation and tree manipulation procedures. Section 8.3 discusses improvements (compared to our previous work [Jeannet et al. 2004]) brought about by the use of an improved meet operation [Arnold et al. 2006]. Section 8.4 discusses experiments to speed up the convergence of the analysis method by injecting likely reachable states at the start nodes of procedures. Section 8.5 compares our method and experimental results with that of Rinetzky et al. [2005b].

All running times were obtained using a 2GHz Pentium M, equipped with 1GB of memory, running Linux.

### 8.1 Analysis of the List-Reversal Example

Given that the input is an acyclic, singly-linked list, the goal of the analysis of the procedure from Figure 1, which destructively reverses an acyclic,

[11] In the limit case, the approach described earlier still solves only *one* equation system: one that is equivalent to phase I of the two-phase approach. Thus, although the results are as precise as the two-phase approach with respect to summary functions, because we do not perform phase II afterwards, the results obtained are imprecise with respect to what phase II discovers about the reachable inputs of callees. In such a case, it would be possible to obtain more accurate information by solving the equations of phase II.



(a) $S_0$ (b) $S_1$ (c) $S_2$

Fig. 17. List-reversal example: The input structure $S_0$ represents all acyclic singly-linked lists of length two or more. The analysis produces the two output structures $S_1$ and $S_2$. (In each structure, unary predicates that have the same non-0 value for all individuals are displayed in the box labeled "Const. unary predicates". The values of the "irrelevant" predicates of the vocabulary are not shown. By convention, the *in*, *tmp*, or *out* qualifier for a predicate whose name includes square-bracket symbols is inserted inside the brackets, for example, $r[n, out, res]$.)

singly-linked list, using recursion to traverse the list, is to show that

(1) the output is an acyclic list
(2) each link of the output list is the reversal of a link of the input list, and vice versa
(3) the cells of the output list are exactly the cells of the input list.

Figure 17 shows how the summary information that we obtain captures the behavior of the recursive list-reversal procedure of Figures 1 and 10. The descriptor of the initial summary transformer at start node $s_{main}$ was the 3-valued structure $S_0$, shown in Figure 17(a), which represents (the identity transformation on) all linked lists of length at least two that are pointed to by program variable `list`. The head of the answer list is pointed to by program variable `res`. At the program's exit node $e_{main}$, the summary transformers were the structures $S_1$ and $S_2$ of Figure 17(b) and Figure 17(c), which represent the transformations that reverse lists of length two, and all lists of length greater than two, respectively.

Note that in both $S_1$ and $S_2$ from Figure 17, each node has the value 0 for the unary predicate $c[n, out]$ and each node has the value 1 for $r[n, out, res]$. This means that no node lies on a directed cycle of `n` fields and all nodes are reachable from the new head of the list `res`, and hence establishes item 1.

As discussed in Section 6.4, relational instrumentation predicates need to be introduced to prevent the loss of essential information. Besides the identity instrumentation predicates defined in Section 6.4, the unary predicates $reverse\_n\_succ[m_1, m_2]$, with $m_1, m_2 \in \{in, out\}$ and $m_1 \neq m_2$, record whether $n[m_2]$ is the reverse of $n[m_1]$. These are defined by

$$reverse\_n\_succ[m_1, m_2](v) = \forall v_1 : (n[m_1](v, v_1) \Rightarrow n[m_2](v_1, v)). \qquad (15)$$

We also provided the following related constraint rules, which allow to deduce a relationship between $n[in]$ and $n[out]$.

$$id\_succ[n, in, tmp](v) \wedge reverse\_n\_succ[tmp, out](v) \Rightarrow reverse\_n\_succ[in, out](v)$$
$$reverse\_n\_succ[in, tmp](v) \wedge id\_pred[tmp, out](v) \Rightarrow reverse\_n\_succ[in, out](v)$$

Note that only the $reverse\_n\_succ[m_1, m_2]$ predicates and the related constraint rules are specific to the list-reversal example. The other predicates that appear in Figure 17 are shape properties that characterize singly-linked lists. (They have been used in previous papers about shape analysis of list manipulation programs; for example, see Sagiv et al. [2002].) For instance, $r[n, out, list](v)$ holds the value 1 for individuals that are reachable from variable list through a chain of $n[out]$ links.

In structures $S_1$ and $S_2$, the values for the predicates $reverse\_n\_succ[m_1, m_2]$, with $m_1, m_2 \in \{in, out\}$ and $m_1 \neq m_2$, show that for each $n$ link $n[in](v_1, v_2)$ at the entry node $s_{main}$, we have an $n$ link $n[out](v_2, v_1)$ at the exit node $e_{main}$. In other words, the procedure reverses all of the $n$ links; this establishes item 2.

Finally, in both of the output structures $S_1$ and $S_2$, we find that $r[n, in, list](v)$ and $r[n, out, res](v)$ hold for each node. This means that no nodes are either lost or gained, and hence the cells of the output list are exactly the cells of the input list; this establishes item 3.

From the previous discussion, it should be clear that the set of 3-valued structures $\{S_1, S_2\}$ establishes the desired properties: the output list is the reversal of the input list, and no elements are either lost or gained.

We generalized this experiment by having procedure `main` call procedure `rev` twice, as in Figure 2(b). To achieve the same level of accuracy as we obtained for a single call on `rev`, we needed to introduce an additional family of unary instrumentation predicates, $reverse\_n\_pred[m_1, m_2]$, whose definition is the same as $reverse\_n\_succ[m_1, m_2]$ (Eq. (15)), except with $v$ and $v_1$ exchanged. With these additional instrumentation predicates, we were able to establish that the second call to `rev` always restores the initial memory configuration.

## 8.2 Experimental Results on Lists and Trees

Tables VI and VII present our experimental results on lists and trees. In these analyses, memory allocation and deallocation is modeled using a pool of free cells [Reps et al. 2003]. The instrumentation predicates related to data structures (lists and trees) are given in Table V and Table VIII. For sorted lists and trees, we introduce the total-order core predicate $leq(v_1, v_2)$ described in Remark 4.3. We also introduce the related predicates of Table IX.

All analyses start with a memory heap consisting of a summary node that represents the free-cell pool and another summary node that represents any context. The core predicate $leq(v_1, v_2)$ evaluates globally to $\mathbf{1/2}$. The examples

Table VI. Experimental Results on Unsorted and Sorted Lists

| | Iterative | | Recursive | |
|---|---|---|---|---|
| Program | # of structs | Time (sec) | # of structs | Time (sec) |
| a. programs on unsorted lists | | | | |
| **create** creates a list of any length | 3/3 | 0.8 | 4/3 | 1 |
| **append** (create) appends 2 lists | 12/9 | 5.5 | 11/9 | 5.5 |
| **split** (create) cuts a list into 2 lists | 7/6 | 2.3 | 7/6 | 2.3 |
| **reverse** (create) destructive list reversal | 9/4 | 4.3 | 5/4 | 2.4 |
| **revappend** (create) reverse-append (using an accumulator parameter) | 12/4 | 4.8 | 15/12 | 6.2 |
| **insert** (create) inserts a cell at a random place in a list | 9/7 | 4.5 | 10/7 | 4.2 |
| **delete** (create) removes a cell at a random place in a list | 9/8 | 5.6 | 14/10 | 4.9 |
| **merge** (create) merges randomly 2 lists<br>**merge*** | | | 92/49<br>32/13 | 92<br>23 |
| **splice** (create) splices 2 lists (specialized merge)<br>**splice*** | | | 10/10<br>9/7 | 13<br>11 |
| b. programs on sorted lists | | | | |
| **create** creates a list of any length | 4/3 | 1 | 4/3 | 1 |
| **append** (create) appends 2 lists | 12/9 | 6 | 11/9 | 6 |
| **split** create) cuts a list into 2 lists | 7/6 | 3 | 7/6 | 3 |
| **reverse** (create) destructive list reversal | 9/4 | 4.8 | 5/4 | 3 |
| **revappend** (create) reverse-append (using an accumulator parameter) | 12/4 | 5.5 | 15/12 | 7 |
| **createo** (inserto) creates a sorted list using **inserto** | 7/3 | 8.5 | 9/3 | 8.5 |
| **inserto** (createo) inserts a cell in the right place in a sorted list | 9/7 | 10 | 11/8 | 10 |
| **deleteo** (createo,inserto) removes a cell with a given key from a sorted list | 188/16 | 114 | 52/23 | 35 |
| **mergeo** (createo,inserto) merges 2 sorted lists in one sorted list<br>**mergeo*** | | | 120/64<br>32/13 | 161<br>45 |
| **spliceo** (createo,inserto) splices 2 sorted lists (interleaves their cells)<br>**spliceo*** | | | 10/10<br>10/7 | 19<br>19 |
| **tailsort** (create, inserto) sorts a list recursively using insert<br>**tailsort*** | | | 110/93<br>23/3 | 135<br>15 |
| **insertionsort** (create, inserto) insertion sort (using an accumulator parameter)<br>**insertionsort*** | | | –<br>65/3 | –<br>35 |
| **mergesort** (create,split,mergeo) mergesort<br>**mergesort*** | | | –<br>69/3 | –<br>113 |

The names in parentheses indicate the other procedures that are analyzed in the example. The stars indicate the introduction of "blurring functions" in dataflow equations. The column "# of structs" indicates (i) the maximum number of logical structures at any control point of the main procedure, and (ii) the maximum number of logical structures at the summary point.

Table VII. Experimental Results on Unsorted and Sorted Trees

| | Recursive | |
|---|---|---|
| Program | # of structs | Time (sec) |
| a. programs on unsorted trees | | |
| **create** creates an unsorted tree of any size (possibly empty) | 10/5 | 13 |
| **create*** | 10/3 | 11 |
| **spliceLeft** (create) inserts a tree as the leftmost child of another tree | 21/11 | 47 |
| **insert*** (create) inserts a cell in a tree | 14/7 | 47 |
| **find*** (create) finds a cell in a tree | 74/18 | 100 |
| **removeRoot** (create,spliceLeft) remove the root of a tree | 12/9 | 63 |
| **remove*** (create,spliceLeft,removeRoot) remove a cell in a tree | 53/25 | 593 |
| **rotate** (create) exchange left and right subtrees of all nodes | 11/5 | 64 |
| b. programs on sorted trees | | |
| **create** creates an unsorted tree | 26/5 | 61 |
| **create*** | 10/3 | 14 |
| **insertu*** (create) inserts a cell in an (unsorted) tree | 14/7 | 59 |
| **spliceLeft** (create) inserts an (unsorted) tree as the leftmost child of another (unsorted) tree | 21/11 | 109 |
| **createo*** (insert) creates a sorted tree | 16/7 | 654 |
| **insert*** (createo) inserts a cell in a sorted tree | 35/12 | 676 |
| **find*** (createo,insert) finds a cell with a given key in a sorted tree | 41/15 | 771 |
| **removeRoot*** (createo,insert,removeRoot,spliceLeft) removes a cell with a given key in a sorted tree | 12/9 | 1160 |
| **remove*** (createo,insert,removeRoot,spliceLeft) removes a cell with a given key in a sorted tree | 30/15 | 1888 |
| **split*** splits a tree into two trees according to a key, one with cells less than the key, one with cells greater than the key | 51/18 | 1780 |
| **rotate** (create) exchange left and right subtrees of all nodes | 19/5 | 101 |

The names in parentheses indicate the other procedures that are analyzed in the example. The stars indicate the introduction of "blurring functions" in dataflow equations. The column "# of structs" indicates (i) the maximum number of logical structures at any control point of the main procedure, and (ii) the maximum number of logical structures at the summary point.

are named according to the main analyzed procedure, but for most of them the main procedure first calls one or more data-structure-creation procedures, and possibly subprocedures, which are also analyzed from scratch.

*Analysis goals.* The goal of each analysis run is to establish that a data-structure invariant is preserved (or reestablished), and that the summary obtained for each procedure captures its effect with sufficient precision. For unsorted lists (respectively, trees), the output should be a well-formed list (respectively, tree), without cell sharing, cycles, and memory leaks. Additionally, for sorted lists (respectively, trees), the output should satisfy shape properties that define the proper ordering of cells in the data structure. The input/output invariant that the summary of a procedure should capture depends on the procedure. Figure 17 shows the procedure summary computed for the list-reversal example, which shows that the output list is composed of exactly the same set of

Table VIII. Defining Formulas of Instrumentation Predicates Related to Binary Trees

| $p$ | Intended Meaning and $\psi_p$ |
|---|---|
| $\textit{down}(v_1, v_2)$ | At least one field of $v_1$ points to $v_2$:<br>$\textit{left}(v_1, v_2) \vee \textit{right}(v_1, v_2)$ |
| $\textit{both}(v_1, v_2)$ | Both fields of $v_1$ points to $v_2$:<br>$\textit{left}(v_1, v_2) \wedge \textit{right}(v_1, v_2)$ |
| $\textit{r\_down}[z](v)$ | Reachability by any field from a variable $z$:<br>$\exists v_1 : z(v_1) \wedge \textit{down}^*(v_1, v)$ |
| $\textit{shared\_down}(v)$ | Shared property:<br>$\exists v_1, v_2 : v_1 \neq v_2 \wedge \textit{down}(v_1, v) \wedge \textit{down}(v_2, v)$ |
| $\textit{cyc\_down}(v)$ | Cyclicity property:<br>$\exists v_1 : \textit{down}(v, v_1) \wedge \textit{down}^*(v_1, v)$ |

Table IX. Defining Formulas of Instrumentation Predicates Related to Ordering of Cells

| $p$ | $\psi_p$ and Intended Meaning |
|---|---|
| Sorted lists: | |
| $\textit{orda}[n](v)$ | The n field of $v$ points to a cell $v_2$ with $v \leq v_2$:<br>$\exists v_2 : n(v, v_2) \wedge \textit{leq}(v, v_2)$ |
| $\textit{ordb}[n](v)$ | The n field of $v$ is null: $\forall v_2 : \neg n(v, v_2)$ |
| $\textit{ord}[n](v)$ | Property of all cells of a sorted list<br>$\textit{orda}[n](v) \vee \textit{ordb}[n](v)$ |
| Sorted binary tree | |
| $\textit{orda\_right}(v)$ | The keys of the right subtree of $v$ are greater than the key of $v$:<br>$\textit{orda\_right}(v) = \exists v_1 : \textit{right}(v, v_1) \wedge \forall v_2 : \textit{down}^*(v_1, v_2) \Rightarrow \textit{leq}(v, v_2)$ |
| $\textit{orda\_left}(v)$ | The keys of the left subtree of $v$ are less than the key of $v$:<br>$\textit{orda\_left}(v) = \exists v_1 : \textit{left}(v, v_1) \wedge \forall v_2 : \textit{down}^*(v_1, v_2) \Rightarrow \textit{leq}(v_2, v)$ |
| $\textit{ordb}[n](v)$ | The n field of $v$ is null: $\forall v_2 : \neg n(v, v_2)$ |
| $\textit{ord\_tree}(v)$ | The tree is sorted:<br>$\textit{ord\_tree}(v) = (\textit{ordb[right]}(v) \vee \textit{orda\_right}(v)) \wedge$<br>$(\textit{ordb[left]}(v) \vee \textit{orda\_left}(v))$ |

cells as the input list, and that for each cell, the incoming $n$ link has become an outgoing $n$ link towards the same cell. For the **insert** and **delete** examples, the summary pinpoints the inserted or deleted cell (see Figure 13(a)). In general, we observed that when the analysis fails to capture a precise approximation of the summary of a procedure, the abstract memory configurations obtained at the return site of the procedure do not establish that the expected data-structure invariants hold.

Note that the shape property that characterizes an ordered tree is much more complex than the shape property that characterizes an ordered list (see Table IX). A list is sorted if and only if each of its cells satisfies a local shape property (namely, that it is in the right order with respect to its immediate successor), whereas a tree is sorted if and only if each of its cells satisfies a global shape property (namely, that it is in the right order with respect to all of its children). The resulting more complex instrumentation predicates for trees explain, for instance, the time difference between the **spliceLeft** example on unsorted trees and on sorted trees. In the latter case, the analyzer must propagate the values of instrumentation predicates that hold information about ordering properties.

The analysis times are quite high for procedures on sorted trees. However, the ability to automatically infer correct summaries for procedures that manipulate sorted trees is a major success for our technique. Indeed, other approaches to interprocedural shape analysis have not yet tackled this challenge. For instance, the tree analyses presented in Rinetzky et al. [2005b] do not establish that orderedness is maintained.

*Choice of appropriate instrumentation predicates.* The instrumentation predicates (Section 5.2.1) that characterize a data structure's shape properties (such as those defined in Tables V, VIII, and IX) are needed for the analysis to infer interesting information. As soon as the data structures manipulated by the analyzed program are large enough to generate summary nodes in abstract structures, these data structures cannot be characterized accurately without these instrumentation predicates.

Concerning relational instrumentation predicates (see Section 6.4), besides the $id\_succ[n, m_1, m_2]$ predicate that is needed to model the identity relationship that holds at the entry of procedures, they are also needed in several procedure summaries to capture crucial information about the before and after states. Section 8.1 discussed the predicate $reverse\_n\_succ[m_1, m_2]$ that models the reversal of $n$ links, used in the analysis of `reverse` and `revappend`. For trees, `rotate` requires a similar relational predicate. The other examples in Tables VI and VII do not require specific relational instrumentation predicates.

The omission of necessary instrumentation predicates quickly leads to useless analysis results: an initial minor loss of precision generally leads to a major loss of precision. The methodology with respect to this issue consists of checking whether the provided instrumentation predicates allow capturing both (i) shape properties that characterize the data structure, and (ii) the effects of the procedures in the analyzed program. We needed some trial-and-error steps to define the appropriate instrumentation predicates for sorted trees.

An alternative approach to the problem of choosing appropriate instrumentation predicates would have been to use the method developed by Loginov et al. [2005] and Loginov [2006] for performing automatic abstraction refinement, using inductive logic programming to identify candidate instrumentation predicates. We did not attempt to use that approach in this work.

*Introduction of "blurring functions" in the analysis.* From the sorted-list examples, one can observe that the analysis time and complexity (in terms of the number of structures representing the summary function) becomes high for merging and sorting procedures. This is due to the fact that our abstraction is sometimes more precise than necessary, and this can cause combinatorial explosion. For instance, for the merge procedure, the abstraction remembers, for each cell of the resulting list, whether it belonged to the first argument list or the second one. The many possible interleavings of the first cells in the resulting lists causes a combinatorial explosion in the result (see Figure 18). This is all the more frustrating because this information is rarely relevant: the properties

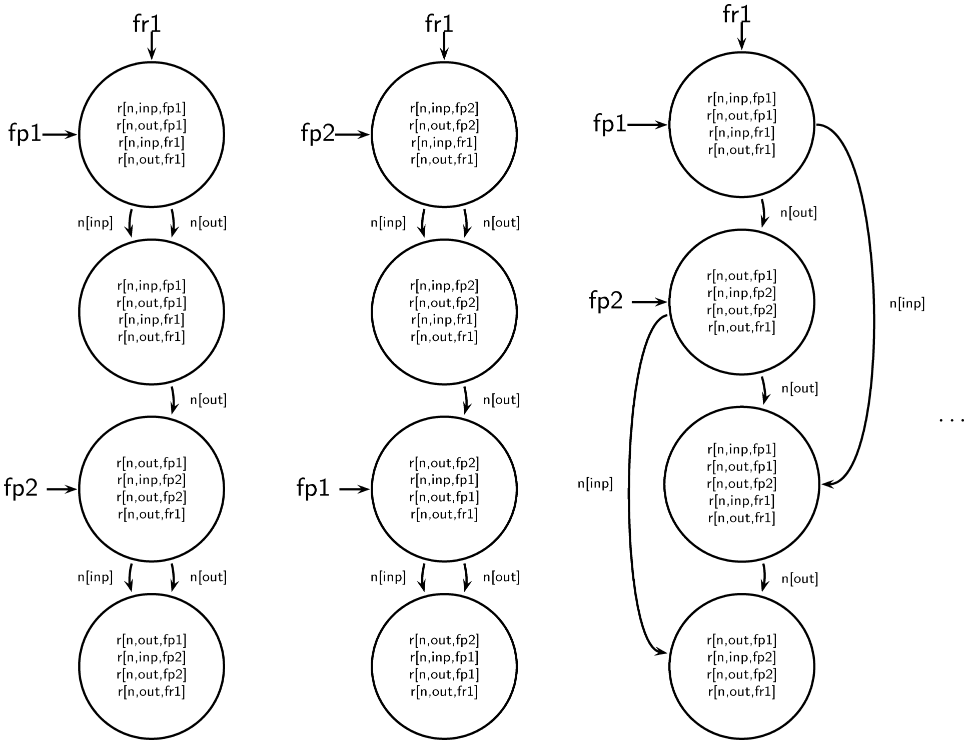


Fig. 18. Combinatorial explosion with the summary of merge: illustration with 2 lists of size 2.

that the summary function of procedure `reverse` should capture accurately are as follows.

(1) Each cell in the result list was a cell in one of the two input lists, and vice versa.

(2) The result is a list, and it is a sorted list.

One way to limit this combinatorial explosion is to apply an extra abstraction step at the end of the procedure. For reverse,

(1) we introduce an instrumentation predicate $r\_n\_\mathit{fp1}or2[m](v) = r[n,m,\mathit{fp1}](v) \vee r[n,m,\mathit{fp2}](v)$ indicating whether a cell is reachable from one of the two list arguments *fp1* and *fp2*;

(2) we forget the value of $n[\mathit{inp}](v_1,v_2)$ and related predicates $r[n,\mathit{inp},\mathit{fp1}](v)$, $r[n,\mathit{inp},\mathit{fp2}](v)$ and $r[n,\mathit{inp},\mathit{fr1}](v)$ on all cells reachable from the result, using the assignment:

$$\begin{aligned}
n[\mathit{inp}](v_1,v_2) &= (r[n,\mathit{out},\mathit{fr1}](v_1)\,?\,1/2 : n[\mathit{inp}](v_1,v_2))\\
r[n,\mathit{inp},\mathit{fp1}](v) &= (r[n,\mathit{out},\mathit{fr1}](v)\,?\,1/2 : r[n,\mathit{inp},\mathit{fp1}](v))\\
r[n,\mathit{inp},\mathit{fp2}](v) &= (r[n,\mathit{out},\mathit{fr1}](v)\,?\,1/2 : r[n,\mathit{inp},\mathit{fp2}](v))\\
r[n,\mathit{inp},\mathit{fr1}](v) &= (r[n,\mathit{out},\mathit{fr1}](v)\,?\,1/2 : r[n,\mathit{inp},\mathit{fr1}](v)).
\end{aligned}$$

The same phenomenon holds for sorting procedures, where the main information to be captured is that the resulting list is a sorted permutation of the input list, but where a (partial) knowledge about the applied permutation is superfluous. The starred versions of the examples in Tables VI and VII refer to versions of the dataflow equations in which such "blurring" functions are introduced to forget information considered irrelevant.

Our methodology with respect to the introduction of blurring functions is related to the choice of instrumentation predicates, with the difference that it is guided only by performance issues. If the existing instrumentation predicates lead to combinatorial explosion with respect to the desired procedure summary, this motivates the application of a blurring function, together with the possible addition of an instrumentation predicate to preserve essential information. Our experience was that adding adequate blurring functions and related instrumentation predicates was quite easy to do, once the origin of the combinatorial explosion was identified (either theoretically or experimentally). The main issue is to blur enough predicates, otherwise some of them might again take on definite values after semantic reduction via Coerce. This is because instrumentation predicates are not independent from each other.

### 8.3 Improvement Brought About by the Meet Operator

Compared to Jeannet et al. [2004], our implementation of interprocedural analysis has been improved by the use of a precise meet operation on the abstract domain of 3-valued structures, proposed by Arnold et al. [2006] and based on graph matching, as mentioned in Section 7.2.

In Jeannet et al. [2004], we used an approximate implementation of the meet of two 3-valued structures, based on the conversion of one of the argument structures to a set of constraint rules, and the application of these additional constraints to the other argument structure using the Coerce and Focus operations, which were briefly described in Section 5.2.2. The approximations came both from the conversion to constraints and the restricted use of the Focus operation. To be exact would require the analysis to focus (temporarily) on all predicates common to the two 3-valued structures; for efficiency reasons we decided to focus only on predicates that represent pointer variables. The method was still rather inefficient.

—The conversion of a 3-valued structure to a 3-valued logical formula and then to constraint rules generates many rules, in particular due to the restricted syntax allowed for such rules in TVLA. Given a 3-valued structure of size $n$ on a vocabulary of size $p_1 + p_2$, where $p_i$ is the number of predicates of arity $i$, the number of generated constraint rules is in $\mathcal{O}(n \cdot p_1 + n^2 \cdot p_2)$.

—The Coerce operation is the most expensive operation in the TVLA implementation.[12]

[12] It should be noted that the inefficiency of the Coerce operation was a general problem in past versions of the TVLA system. It motivated the work reported in Arnold [2006], which obtained substantial speedups by replacing pairs of Focus/Coerce operations by the meet operation whenever possible. It also motivated the work of Bogudlov et al. [2007a, 2007b] who developed techniques that

Table X. Comparison of the Use of Resources with the Old and New Meet Operators

| | Old meet | | New meet | |
|---|---|---|---|---|
| Program | # of structs | Time (sec) | # of structs | Time (sec) |
| programs on unsorted lists (recursive version) | | | | |
| **reverse** destructive list reversal | 7 | 4.7 | 5/4 | 2.4 |
| **insert** inserts a cell at a random place in a list | 23 | 82 | 10/7 | 4.2 |
| **delete**<br>removes a cell at a random place in a list | 32 | 84 | 14/10 | 4.9 |

Analysis with the old meet does not include the creation of input lists. It also requires 2 additional instrumentation predicates for the **insert** and **delete** examples, due to the approximation induced by the meet.

Table XI. Interprocedural Analysis Method

| | Normal | | Accelerated | |
|---|---|---|---|---|
| Program | # of structs | Time (sec) | # of structs | Time (sec) |
| a. programs on sorted lists | | | | |
| **tailsort*** (create,insert) | 23/3 | 15 | 18/3 | 8.8 |
| **insertionsort*** (create,insert) | 65/3 | 35 | 65/3 | 27 |
| **mergesort*** (create,split,merge) | 69/3 | 113 | 48/3 | 40 |
| b. programs on sorted trees | | | | |
| createo* (insert) | 16/7 | 654 | 14/8 | 250 |

The gains obtained by the use of the precise meet operation of Arnold et al. [2006] are illustrated in Table X for a few simple examples. The gain in efficiency is impressive, but the gain in precision is also important: for the **insert** and **delete** examples, we did not need to introduce specific instrumentation predicates to capture the effect of these procedures (the triangular pattern shown in Figure 13(a)). This precision issue prevented us from experimenting with the old meet implementation on our full set of examples.

## 8.4 Speeding Up the Analysis by Modifying the Equations

In Section 7.3, we discussed the possibility of speeding up the convergence of the analysis by injecting (a subset of) likely reachable states at the start nodes of procedures, which may reduce the number of iteration steps needed for reaching a fixpoint.

We experimented with this technique on programs that consist of several recursive procedures: we injected the set of all well-formed (and possibly ordered) lists at the entry of all list-manipulating procedures. The results are given in Table XI, and show that in such cases this technique is very efficient. Note that in the examples from Table XI, all procedures are recursive, which leads to more complex dependences than in the example shown in Figure 16.

allowed Coerce to run over an order of magnitude faster. These techniques were not incorporated into the version of TVLA that implements the methods described in the present article. The methods of Bogudlov et al. [2007a, 2007b] are essentially orthogonal to the ones that we developed, and thus the speedups that would be obtained by incorporating their techniques into our implementation should be comparable to the speedups reported in Bogudlov et al. [2007a, 2007a].

```
void main(){
  List list = create();
  List acc = NULL;
  List res = insertionsort(list,acc);
}
List insertionsort(List list, List acc){
  List res,t,tt;
  if (list==NULL)
    res = acc;
  else {
    t = list->n; list->n = NULL;
    tt = insert(acc,list);
    res = insertionsort(t,tt);
  }
  return res;
}
```

```
List insert(List list, List cell){
  List res;
  if (list!=NULL && cell->key > list->key){
    t = list->n; list->n = NULL;
    res = insert(t,cell);
    list->n = res;
    res = list;
  }
  else {
    if (list==NULL) cell->n = NULL;
    else cell->n = list;
    res = cell;
  }
  return res;
}
```

Fig. 19. Insertionsort example.

For the **insertionsort** example depicted on Figure 19, with the standard technique, **insertionsort**, and then **insert**, will first be called with the `acc=NULL` argument. **insert** will be analyzed for this base case, then **insertionsort** will be called with a one-element list in `acc`, which will be propagated later in the body of **insert**. So it takes several steps to infer that **insert** might be called with any sorted list. Injecting the set of all sorted lists at the entry of **insert** allows to compute quicker the complete summary of **insert**, which also induces a faster propagation of the call to **insert** in **insertionsort**.

### 8.5 Comparison with Cutpoint Semantics and Tabulated Representation

In this section, we compare our method and experimental results with those of Rinetzky et al. [2005b]. The two methods are built on the same abstract domain of 3-valued logical structures, and both implement a context- and flow-sensitive interprocedural shape analysis based on procedure summarization.

The effective reuse of procedure summaries in different calling contexts motivated the development of mechanisms to allow parts of the heap that are not relevant to the procedure's actions to be ignored [Rinetzky et al. 2005b]. Rinetzky et al. [2005b] use a tabulated representation, that is, using pairs of abstracted structures, rather than abstractions of paired structures, to capture summaries of procedures, and the notion of *cutpoints* is used to eliminate details of the heap that are inessential to the callee, thereby permitting procedure summaries to be used in different calling contexts.[13] In Section 9, we describe the similarities and differences between the methods more thoroughly, and discuss how a similar effect of eliminating details is obtained essentially for free with our approach.

Table XII compares the two analyses on a set of examples. We followed Rinetzky et al. [2005b] by not analyzing procedures in isolation, but instead

[13] When control is passed from the caller to the callee, the cutpoints represent the frontier of vertices in the part of the heap visible to the callee that are reachable from the caller's pointer variables (or from pointer variables of other procedures further back in the stack). During the execution of the callee, it is necessary to track all nodes that are reachable from the local variables, the global variables, and the cutpoints, but other parts of the heap structure can be removed.

Table XII. Times for Cutpoint-Based Analysis vs. Relational Analysis

|  | Cutpoint | Relational |  |
|---|---|---|---|
| Program | -based | std. | * |
| a. (Sorted) list-manip. programs |  |  |  |
| create | 8 | 8 | 8 |
| insert | 46 | 10 | 10 |
| delete | 46 | 35 | 35 |
| reverse | 32 | 3 | 3 |
| revappend | 47 | 7 | 7 |
| merge | 83 | 161 | 45 |
| insertionsort | 265 | >1000 | 35 |
| tailsort | 65 | 135 | 15 |
| mergesort | 576 | >1000 | 113 |

|  | Cutpoint | Relational |  |
|---|---|---|---|
| Program | -based | std. | * |
| b. (Unsorted) tree-manip. programs |  |  |  |
| create | 10 | 61 | 14 |
| insert | 25 | — | 47 |
| find | 67 | — | 100 |
| removeRoot | 49 | — | 63 |
| remove | 114 | — | 593 |
| spliceLeft | 26 | — | 47 |
| rotate | 43 | — | 64 |

All times are in seconds. (The columns labeled * report the times for analysis runs in which blurring functions are applied. A long dash (—) means that the run was not attempted.)

analyzing a full program from scratch. This means that the analysis time for the mergesort example includes the analysis time for the creation of a list (**create**), as well as the auxiliary procedures (**split** and **merge**). Both analyses were executed on the same computer, using the same version of the TVLA system (with the exception of the additions mentioned at the very beginning of this section).

*Sorted-list examples.* For this set of examples, the relational method is generally as efficient as, or more efficient than, the cutpoint-based method. It should be noted, however, that the relational method sometimes requires the application of blurring functions to obtain reasonable performance, but in such cases the gain in performance is significant, even with respect to the cutpoint-based analysis. The latter is somewhat surprising because the cutpoint-based analysis tabulates pairs of structures, and, as discussed in Section 6.2 and illustrated in Figure 13, the information computed by the relational analysis is much more precise than the information that is computed when a tabulated representation is used.

*Unsorted-tree examples.* Because Rinetzky et al. [2005b] did not try to analyze examples with sorted trees, the experiments dealt only with unsorted trees: the ordering relation between tree cells is abstracted away. The execution times are better for the cutpoint-based method, but remain of the same order of magnitude, with the exception of the **remove** example. The latter example demonstrates that the advantages of the relational analysis in terms of precision can have a cost in terms of efficiency, even when blurring functions (Section 8.2) are applied. In the case of the **remove** procedure, the extra precision of the relational analysis causes the number of cases that the analyzer has to consider to increase: in particular, the set of output two-vocabulary structures at the exit node of **remove** (i.e., the procedure summary for **remove**) relates an output tree, in which a cell has been removed, to an input tree, which contains the cell. Consequently, for essentially the same output tree, the analyzer ends up enumerating a number of different two-vocabulary structures according to the different possible positions in the input tree of the cell to be removed: the cell to be removed is the root; the cell to be removed is a left or right child of the root; or the cell to be removed is one that lies deeper in the left or right subtree of the root cell.

## 9. RELATED WORK

*General approaches to interprocedural analysis.* One can distinguish two main approaches to interprocedural static analysis. The first approach, called the *functional approach* (after the name used in Sharir and Pnueli [1981]), uses a denotational semantics of the analyzed program and consists of two steps. The first step computes predicate transformers associated with the procedures of the program by finding a fixpoint of a set of equations over predicate transformers. The operations used in these equations are (primarily) transformer composition and transformer join. The second step (repeatedly) applies a composed predicate transformer for a program path to some predicate that characterizes the possible input states, to obtain a predicate that holds at the end of the path. Cousot and Cousot [1977], Sharir and Pnueli [1981], and Knoop and Steffen [1992] apply this approach to different classes of programs.

The second approach, which we call the *operational approach*, adopts an operational semantics for programs. Here, as in many intraprocedural verification techniques, the predicates are propagated along the edges of the program's control-flow graph, using the predicate transformers associated with program statements and conditions, until a fixpoint is reached. The analysis can be viewed as a symbolic execution of the program in which values are replaced by properties. In contrast with the functional approach, there is no computation of (composed) predicate transformers associated with blocks of instructions or procedures. However, to simulate the execution of the program, one needs to take into account the program's call stack: when a procedure returns to its caller, the call site should be popped from the stack and the local state of the caller should be restored to the state that it had before the call. The "call-strings" approach of Sharir and Pnueli [1981] provides one way to address this issue, by maintaining additional information in the abstract domain to overapproximate the state of the call stack.

Techniques based on pushdown systems [Bouajjani et al. 1997; Finkel et al. 1997] and weighted pushdown systems [Bouajjani et al. 2003; Reps et al. 2005] contain elements of both the functional and operational approaches. Jeannet and Serwe [2004] show how the functional and operational approaches can be derived as an abstract interpretation of the standard operational semantics, modeled using a stack of activation records. Once the interprocedural semantics is defined in this way, a second abstraction step may be used to abstract the data (in our case, the values of variables and linked memory cells). This is the approach we followed in Section 7.1, with the variations described in Section 7.3.

*Interprocedural shape analysis.* Several other papers have studied interprocedural shape analysis using canonical abstraction. In Rinetzky and Sagiv [2001], the store is augmented to include the runtime stack as an explicit data structure. The storage abstraction used in Rinetzky and Sagiv [2001] is an abstraction of the store augmented in this fashion. In essence, the collection of activation records that form the stack are abstracted using an abstraction for linked lists. This "stack materialization" approach causes certain technical complications; they are not insurmountable, but do cause the designer of an abstract interpretation to have to identify certain shape properties that relate the state of the stack and the state of the heap during the execution of the program (in particular, how the heap cells reachable from the visible and invisible instances of local variables are related). This approach is reminiscent of the "call-strings" approach; in contrast, the approach used in the present article was inspired by the functional approach, in which the stack is not materialized as an explicit data structure; instead it is an implicit part of the programming-language semantics. Thus, the designer of an abstract interpretation does not need to be concerned with the "shape" of the runtime stack nor with such things as visible and invisible instances of local variables. Because of the different nature of the information obtained, Rinetzky and Sagiv [2001] can only show that a list reversed twice yields a list with the same head and the same set of memory cells (in some order) as the initial list, while our method shows that it yields the same initial list.

As mentioned in Section 8.5, Rinetzky et al. [2005b] implement a context- and flow-sensitive analysis that is also inspired by the functional approach, but which uses tabulation to represent the summaries of procedures. The effective reuse of procedure summaries in different calling contexts is made possible by using the notion of *cutpoints* and by considering cutpoint-free programs.[14] As for Rinetzky and Sagiv [2001], the resulting analysis is less precise than ours because their tabulated representation is less expressive than our relational representation, in which one can track (an approximation of) the evolution of individual objects. It may perform more efficiently, particularly on trees, but it has not yet been applied to ordered trees, where the ingredients of invariants satisfied by ordered trees need to be tracked. Our approach has the benefit of generality (it is not restricted to cutpoint-free programs) and conceptual simplicity: it reuses the same algorithms as the intraprocedural analysis, and relational composition is performed using the standard notions of intersection and elimination.

A method for performing interprocedural shape analysis using procedure specifications and assume-guarantee reasoning is presented in Yorsh et al. [2004]. There, it is assumed that a specification for each procedure, a pre- and postcondition, is already known; the technique presented in Yorsh et al. [2004] can be used to interpret a procedure's pre- and postcondition in the most precise way (for a given abstraction). For every procedure invocation, one checks if the current abstract value potentially violates the precondition; if it does, a warning is produced. At the point immediately after the call, one can assume that the postcondition holds. Similarly, when a procedure is analyzed, the precondition is assumed to hold on entry, and at end of the procedure the postcondition is checked. The work described in the present article is complementary to Yorsh

[14] In cutpoint-free programs [Rinetzky et al. 2005a], the nodes pointed to by a caller's parameters always dominate the nodes that are reachable from the caller's pointer variables (or from pointer variables of other procedures further back in the stack).

et al. [2004]: our work provides a way to identify procedure specifications (in the form of sets of 2-vocabulary 3-valued structures) that can be used with the method from Yorsh et al. [2004].

Several techniques have been suggested for automatically checking the partial correctness of programs annotated with loop invariants and pre- and postconditions [Møller and Schwartzbach 2001; Berdine et al. 2005; Lahiri and Qadeer 2008]. Compared to our approach to shape analysis, those techniques can be faster; in particular, annotations can drastically reduce the cost of interprocedural shape analysis because they allow the correctness of a set of procedures to be checked modularly, using a linear pass over each procedure's body. However, the burden of requiring programmers to express loop invariants and the required pre- and postconditions is much higher than the effort required for providing adequate instrumentation predicates in our method.

A recent approach to interprocedural shape analysis is based on separation logic, which has been designed for performing context-independent reasoning about memory shapes [Gotsman et al. 2006]. It has to take care of cutpoints, and to abstract them if too many cutpoints appear in the course of the analysis. In our method, pointer variables to cutpoints in the caller are forgotten in the callee, but are recovered upon return from the callee thanks to the meet operation in Eq. (14). Cutpoints were also used to develop interprocedural shape-analysis algorithms that are not based on canonical abstractions [Marron et al. 2008]. We believe that the principles that underlie our relational analysis (i.e., the use of abstractions of two-vocabulary structures) are also applicable for other abstractions as long as they support the right interface operations (e.g., projection and meet).

*"Heap modularity"*. Both Rinetzky et al. [2005b] and Gotsman et al. [2006] state that their techniques are fully "heap modular" in the sense that the procedure summaries computed by the analyses deal only with the reachable parts of the heap and ignore the (unreachable) context of the caller in the callee, which cannot be modified by the callee.

This effect is obtained naturally with our approach. Because most core and instrumentation predicates are related to reachability from visible variables, the part of the heap that is not reachable from the local variables in a callee is summarized with a single (or a few) "context" summary nodes. When the callee returns to its caller, this context summary node is materialized again by the meet of the summary relation at the return site of the callee and the relation at the call site. In fact, because some predicates are independent of reachability properties (predicates related to cyclicity or sharing), there may be several context summary nodes. In such cases, at the entry of the callee, a predicate-update formula (refer to Section 4.3) may be used to assign the value $\mathbf{1/2}$ to those predicates for unreachable cells, as follows.

$$p'(v) = \mathit{reachable\_from\_input\_parameters}(v)\ ?\ p(v)\ :\ 1/2$$

This induces a more effective merging of abstract cells not reachable in the callee (hence not modifiable by the callee). The information is recovered during the processing at the procedure return site.

*Abstract transformers.* The analysis described in this article uses 3-valued structures over a doubled vocabulary. A similar approach is standard when concrete transition relations are expressed by means of formulas. For instance, the semantics of a statement `x := y+1` can be expressed as $(x' = y + 1) \wedge (y' = y)$. Statements such as `x := y+1` can be transformed into composable abstract transformers for programs that manipulate numeric data, using several numeric lattices (e.g., polyhedra [Cousot and Halbwachs 1978], octagons [Miné 2006], etc.). A key feature of the approach described in the present article is that relational instrumentation predicates can refer to both the $\mathcal{P}[inp]$ and $\mathcal{P}[out]$ vocabularies. For instance, the family of unary predicates $reverse\_n\_succ[m_1, m_2]$ discussed in Section 8 (with $m_1, m_2 \in \{inp, out\}$ and $m_1 \neq m_2$) records whether $n[m_2]$ is an inverse of $n[m_1]$.

The classic functional approach of Sharir and Pnueli [1981] uses function composition for all operations. As is typically done in analyses based on the numerical abstract domain [Cousot and Halbwachs 1978; Miné 2006], the approach taken in this article might be more properly described as a *hybrid* approach.

(1) Intraprocedural propagation is based on a form of transformer application, rather than transformer composition. That is, for an intraprocedural propagation with respect to transformer $\tau$, the actions of $\tau$ are applied to the second vocabulary, with the first vocabulary kept constant.

(2) Interprocedural propagation is based on the composition of two-vocabulary structures (using three-vocabulary structures, structure meet, and vocabulary projection).

For shape analysis, the advantage of the hybrid approach has to do with the maintenance of instrumentation predicates that express reachability properties. The application step used in item (1) is satisfactory when there are unit-size changes to core relations: the instrumentation-predicate-maintenance formulas created by finite differencing [Reps et al. 2003] are generally able to maintain definite values for instrumentation predicates that express reachability properties for *unit-size changes* to core predicates. The (approximate) composition step used in item 2 generally allows definite values to be retained under the nonunit-size changes to core predicates that occur when applying a procedure summary.

## ACKNOWLEDGMENTS

We are grateful to V. Kuncak for several discussions about the use of two-vocabulary structures in shape analysis; to N. Rinetzky for many discussions about interprocedural shape-analysis methods, as well as for his help with the experiments that compare our methods with his; and to G. Arnold for his help incorporating his work on the meet operation into our implementation.

## REFERENCES

ARNOLD, G. 2006. Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In *Proceedings of the Static Analysis Symposium (SAS'06)*. Lecture Notes in Computer Science, vol. 4134, Springer.

Arnold, G., Manevich, R., Sagiv, M., and Shaham, R. 2006. Combining shape analyses by intersecting abstractions. In *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'06).* Lecture Notes in Computer Science, vol. 3855, Springer.

Ball, T. and Rajamani, S. 2001. Bebop: A path-sensitive interprocedural dataflow engine. *Prog. Anal. Softw. Tools Engin*, 97–103.

Berdine, J., Calcagno, C., and O'Hearn, P.W. 2005. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the Symposium on Formal Methods for Components and Objects (FMCO'05)*. Lecture Notes in Computer Science, vol. 4111, Springer, 115–137.

Bogudlov, I., Lev-Ami, T., Reps, T., and Sagiv, M. 2007a. Revamping TVLA: Making parametric shape analysis competitive. Tech. rep. TR-2007–01-01, Tel-Aviv University, Tel-Aviv, Israel.

Bogudlov, I., Lev-Ami, T., Reps, T., and Sagiv, M. 2007b. Revamping TVLA: Making parametric shape analysis competitive (tool paper). In *Proceedings of the International Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 4590, Springer.

Bouajjani, A., Esparza, J., and Maler, O. 1997. Reachability analysis of pushdown automata: Application to model checking. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'97).* Lecture Notes in Computer Science, vol. 1243, Springer, 135–150.

Bouajjani, A., Esparza, J., and Touili, T. 2003. A generic approach to the static analysis of concurrent programs with procedures. In *Principles of Programming Languange*, ACM Press, New York, 62–73.

Clarke, Jr., E., Grumberg, O., and Peled, D. 1999. *Model Checking*. The MIT Press.

Cousot, P. and Cousot, R. 1977. Static determination of dynamic properties of recursive procedures. In *Formal Descriptions of Programming Concepts*, E. Neuhold, Ed. North-Holland, 237–277.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear constraints among variables of a program. In *Principles of Programming Language*, ACM Press, New York, 84–96.

Finkel, A., B. Willems, and Wolper, P. 1997. A direct symbolic approach to model checking pushdown systems. *Electron. Notes Theor. Comput. Sci. 9*.

Gopan, D., DiMaio, F., N.Dor, Reps, T., and Sagiv, M. 2004. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 2988, Springer, 512–529.

Gotsman, A., Berdine, J., and Cook, B. 2006. Interprocedural shape analysis with separated heap abstractions. In *Proceedings of the Static Analysis Symposium (SAS'06).* Lecture Notes in Computer Science, vol. 4134, Springer, 240–260.

Gries, D. 1981. *The Science of Programming*. Springer.

Jeannet, B., Loginov, A., Reps, T., and Sagiv, M. 2004. A relational approach to interprocedural shape analysis. In *Proceedings of the Static Analysis Symposium (SAS'04).* Lecture Notes in Computer Science, vol. 3148, Springer.

Jeannet, B. and Serwe, W. 2004. Abstracting call-stacks for interprocedural verification of imperative programs. In *Proceedings of the Workshop on Algebraic Methodology and Software Technology (AMAST'04).* Lecture Notes in Computer Science, vol. 3116, Springer.

Knoop, J. and Steffen, B. 1992. The interprocedural coincidence theorem. In *Computing Construction*. Lecture Notes in Computer Science, vol. 641, Springer, 125–140.

Lahiri, S. K. and Qadeer, S. 2008. Back to the future: Revisiting precise program verification using smt solvers. In *Principles of Programming Language*. ACM Press, New York.

Lev-Ami, T., Reps, T., Sagiv, M., and Wilhelm, R. 2000. Putting static analysis to work for verification: A case study. In *Proceedings of the International Symposium on Software Testing and Analysis*. 26–38.

Lev-Ami, T. and Sagiv, M. 2000. TVLA: A system for implementing static analyses. In *Proceedings of the Static Analysis Symposium (SAS'00)*. Lecture Notes in Computer Science, vol. 1824, Springer, 280–301.

Loginov, A. 2006. Refinement-based program verification via three-valued-logic analysis. Ph.D. thesis, Tech. rep. 1574. Computer Science Department, University of Wisconsin, Madison, WI.

Loginov, A., Reps, T., and Sagiv, M. 2005. Abstraction refinement via inductive learning. In *Proceedings of the International Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 3576, Springer.

Manna, Z. and Pnueli, A. 1995. *Temporal Verification of Reactive Systems: Safety*, Springer.

Marron, M., Hermenegildo, M. V., Kapur, D., and Stefanovic, D. 2008. Efficient context-sensitive shape analysis with graph based heap models. In *Computer Construction*. Lecture Notes in Computer Science, vol. 4959, Springer, 245–259.

Miné, A. 2006. The octagon abstract domain. *Higher-Order Symb. Comput. 19*, 1, 31–100.

Møller, A. and Schwartzbach, M. I. 2001. The pointer assertion logic engine. In *Programming Language Design and Implementation*. 221–231.

Reps, T., Horwitz, S., and Sagiv, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Programming Language*. ACM Press, New York, 49–61.

Reps, T., Sagiv, M., and Loginov, A. 2003. Finite differencing of logical formulas for static analysis. In *Proceedings of the European Symposium on Programming*. Lecture Notes in Computer Science, vol. 2618, 380–398.

Reps, T., Schwoon, S., Jha, S., and Melski, D. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program. 58*, 1–2, 206–263.

Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., and Wilhelm, R. 2005a. A semantics for procedure local heaps and its abstraction. In *Proceedings of the 32nd ACM SIGPLAN – SIGACT Symposium on Principles of Programming Languages (POPL'05)*.

Rinetzky, N. and Sagiv, M. 2001. Interprocedural shape analysis for recursive programs. In *Computer Construction*. Lecture Notes in Computer Science, vol. 2027, Springer, 133–149.

Rinetzky, N., Sagiv, M., and Yahav, E. 2005b. Interprocedural shape analysis for cutpoint-free programs. In *Proceedings of the Static Analysis Symposium (SAS'05)*. Lecture Notes in Computer Science, vol. 3672, Springer.

Sagiv, M., Reps, T., and Horwitz, S. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci. 167*, 131–170.

Sagiv, M., Reps, T., and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst. 24*, 3, 217–298.

Schwoon, S. 2002. Model-checking pushdown systems. Ph.D. thesis, Technical University of Munich, Munich, Germany.

Sharir, M. and Pnueli, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, NJ. Chapter 7, 189–234.

Yorsh, G., Reps, T., and Sagiv, M. 2004. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 2988, Springer, 530–545.

Received May 2008; revised December 2008; accepted May 2009