A Relational Approach to Interprocedural Shape Analysis

Bertrand Jeannet[1], Alexey Loginov[2], Thomas Reps[2], and Mooly Sagiv[3]

[1] IRISA; Bertrand.Jeannet@irisa.fr
[2] Comp. Sci. Dept., University of Wisconsin; {alexey,reps}@cs.wisc.edu
[3] School of Comp. Sci., Tel Aviv University; msagiv@post.tau.ac.il

Abstract. This paper addresses the verification of properties of imperative programs with recursive procedure calls, heap-allocated storage, and destructive updating of pointer-valued fields—i.e., *interprocedural shape analysis*. It presents a way to harness some previously known approaches to interprocedural dataflow analysis—which in past work have been applied only to much less rich settings—for interprocedural shape analysis.

1 Introduction

This paper 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 paper builds on past work by several of the authors on static analysis based on 3-valued logic [1, 2] and its implementation in the TVLA system [3]. 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 over-approximation 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 fixed-point-finding procedure on a collection of equations in which each variable in the equation set has a transformer-valued value—i.e., 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 [4–9]. 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 as sets of pairs of input/output values [7, 8, 10]. However, for interprocedural shape analysis, this approach fails to produce useful information. The 3-valued-logic approach to shape analysis is a *storeless* one: 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 is not promising: the representation provides only a weak characterization of a procedure's net effect.

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

Observation 1. *By using* 3*-valued structures over a doubled vocabulary* $\mathcal{P} \uplus \mathcal{P}'$*, where* $\mathcal{P}' = \{p' \mid p \in \mathcal{P}\}$ *and* $\uplus$ *denotes disjoint union, one obtains 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 abstraction 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. Moreover, by extending the abstract domain of 3-valued logical structures [1] with some new operations, it is possible to perform abstract interpretation of call and return statements without losing too much precision (see §4). 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.

Context-sensitive interprocedural shape analysis was also studied in [11]. A major difference is that [11] augments the store to include the runtime stack as an explicit data structure (an idea proposed in [12, 13]); the storage abstraction used in [11] is an abstraction of the store augmented in this fashion. In contrast, in our work the stack is not materialized as an explicit data structure; our approach is based on the creation of summary transformers, in the style of [4–6].

The contributions of our work include the following:

- It provides a method to create a *summary transformer* for each procedure P, which can be used at each call site at which P is called.
- Our analysis obtains more general information than that obtained in [11]:
  - In [11], the result of the analysis for the exit node e_P of a procedure P is (an approximation of) the reachable memory configurations that can arise at e_P
  - In this paper, the result for e_P is (an approximation of) the *relation* between the input memory configurations at the start node s_P of P and the configurations at e_P, restricted to the memory configurations that are reachable at s_P.

  Because of the different nature of the information obtained, our analysis is able to verify that reversing a list twice restores the original list, whereas the method of [11] would only show that it yields a list with the same head and the same set of memory cells (in some order).

– We have been able to apply our method successfully to a richer set of programs. In particular, [11] only studied how to perform interprocedural analysis for recursive *list*-manipulation programs. The method described in this paper is capable of handling certain 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 non-recursive form.)

The remainder of the paper is organized as follows: §2 describes the features of the language to which our analysis applies. §3 reviews the abstract domain of 3-valued logical structures [1]. §4 describes how abstractions of logical structures over a doubled vocabulary are used to create summary transformers and perform interprocedural analysis. §5 discusses experimental results. §6 discusses related work.

2 Programs and Memory Configurations

The analysis applies to programs written in an imperative programming language in which (i) it is forbidden to take the address of a local variable, global variable, or parameter; and (ii) parameters are passed by value. These two features prevent direct aliasing among variables; thus, only heap-allocated structures can be aliased. (Both JAVA and ML follow these conventions.) The running example used in the paper is the list-reversal program of Fig. 1.

```
typedef struct node{ List rev(List x){
  struct node *n;      List y, z;
  int data;            z = x->n;
} *List;               x->n = NULL;
                       if (z != NULL){
List res;                y = rev(z);
void main(List l){       z->n = x;
  res = rev(l);        }
}                      else y = x;
                       return y;
                     }
```

Fig. 1. Recursive list-reversal program.

2.1 Program Syntax

A *program* is defined by a set of procedures P_i, $0 \leq i \leq K$. Each procedure has a set of local variables, and has a number of formal *input parameters* and *output parameters*. To simplify our notation, we will assume that each procedure has only *one* input (resp. output) parameter and *one* local variable; the generalization to multiple parameters and local variables 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* $P_i = \langle \mathsf{fpi}_i, \mathsf{fpo}_i, \mathsf{loc}_i, G_i \rangle$ is defined by its input parameter fpi_i, its output parameter fpo_i, its local variable loc_i, and G_i, its intraprocedural control flow graph (CFG).

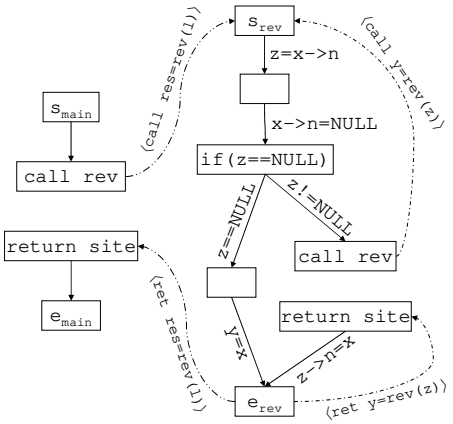


Fig. 2. Interprocedural CFG of the list-reversal program.

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 other nodes of a CFG represent individual statements and branches of a procedure in the usual way,[4] except that a procedure call is represented by two nodes, 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: (i) a *call-to-start* edge from c to the start node of the called procedure, and (ii) 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.

2.2 Representing Memory Configurations with Logical Structures

As in the static-analysis framework defined in [1], concrete memory configurations—or *stores*—are modeled by logical structures. A logical structure is associated with a vocabulary of predicate symbols (with given arities): $\mathcal{P} = \{eq, p_1, \ldots, p_n\}$ is a finite set of predicate symbols, where $\mathcal{P}_k$ denotes the set of predicate symbols of arity k (and $eq \in \mathcal{P}_2$). A logical structure supplies a predicate for each of the vocabulary's predicate symbols. A concrete store is modeled by a 2-valued logical structure for a fixed vocabulary of *core predicates*, $\mathcal{C}$. Core predicates are part of the underlying semantics of the language to be analyzed; they record atomic properties of stores. For instance, Tab. 1 lists the predicates that would be used to represent the stores manipulated by programs that use type `List` from Fig. 1, such as the store shown in Fig. 3. 2-valued logical structures then represent memory configurations: the individuals are the set of memory cells; a nullary predicate represents a Boolean variable of the program; a unary predicate represents either a pointer variable or a Boolean-valued field of a record; and a binary predicate represents a pointer field of a record.[5]

Predicate	Intended Meaning
$eq(v_1, v_2)$	Do v_1 and v_2 denote the same memory cell?
$q(v)$	Does pointer variable `q` point to memory cell v?
$n(v_1, v_2)$	Does the `n`-field of v_1 point to v_2?
$dle(v_1, v_2)$	Is the `data`-field of $v_1 \leq$ the `data`-field of v_2?

Table 1. Core predicates used for representing the stores manipulated by programs that use type `List`. (We write predicate names in *italics* and code in `typewriter` font.)

The 2-valued structure S, shown in the left-hand side of Fig. 4, encodes the store of Fig. 3. S's four individuals, u_1, u_2, u_3, and u_4, represent the four list cells.

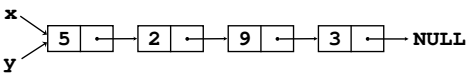


Fig. 3. A possible store, consisting of a four-node linked list pointed to by `x` and `y`.

[4] Alternatively, nodes can represent basic blocks.

[5] To simplify matters, our examples do not involve modeling numeric-valued variables and numeric-valued fields (such as `data`). It is possible to do this by introducing other predicates, such as the binary predicate dle (which stands for "`data` less-than-or-equal-to") listed in Tab. 1; dle captures the relative order of two nodes' `data` values. Alternatively, numeric-valued entities can be handled by combining abstractions of logical structures with previously known techniques for creating numeric abstractions [14].

The following graphical notation is used for depicting 2-valued structures:

- An individual is represented by a circle with its name inside.
- A unary predicate p is represented by having a solid arrow from p to each individual u for which $p(u) = 1$, and by the absence of a p-arrow to each individual u' for which $p(u') = 0$. (If predicate p is 0 for all individuals, p is not shown.)
- A binary predicate q is represented by a solid arrow labeled q between each pair of individuals u_i and u_j for which $q(u_i, u_j) = 1$, and by the absence of a q-arrow between pairs u'_i and u'_j for which $q(u'_i, u'_j) = 0$.

Thus, in structure S, pointer variables x and y point to individual u_1, whose n-field points to individual u_2; pointer variable z does not point to any individual.

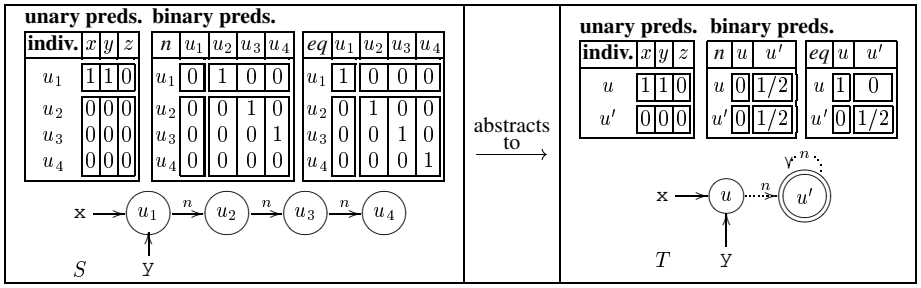


Fig. 4. The abstraction of 2-valued structure S to 3-valued structure T when we use $\{x, y, z\}$-abstraction.

Often we only want to use a restricted class of logical structures to encode stores. To exclude structures that do not represent admissible stores, integrity constraints can be imposed. For instance, the predicate $x(v)$ of Fig. 4 captures whether pointer variable x points to memory cell v; x would be given the attribute "unique", which imposes the integrity constraint that $x(v)$ can hold for at most one individual in any structure.

The concrete operational semantics of a programming language is defined by specifying a structure transformer for each kind of edge e that can appear in a control-flow graph. Formally, the structure transformer τ_e for edge e is defined using a collection of *predicate-update formulas*, $c(v_1, \ldots, v_k) = \tau_{c,e}(v_1, \ldots, v_k)$, one for each core predicate c (e.g., see [1]). These formulas define how the core predicates of a logical structure S that arises at the source of e are transformed by e to create a logical structure S' at the target of e; they define the value of predicate c in S' as a function of c's value in S. Edge e may optionally have a *precondition formula*, which filters out structures that should not follow the transition along e. (In Fig. 2, edges are labeled with statements and conditions of the programming language, rather than with such collections of predicate-update formulas.)

The set of all 2-valued structures over vocabulary $\mathcal{P}$ is denoted by $\mathcal{S}_2[\mathcal{P}]$.

3 The Abstract Domain of 3-Valued Logical Structures

To create abstractions of 2-valued logical structures (and hence of the stores that they encode), we use the related class of 3-valued logical structures over the same vocabulary. In 3-valued logical structures, a third truth value, denoted by $1/2$, is introduced to denote uncertainty: in a 3-valued logical structure, the value $p(\vec{u})$ of predicate p on a tuple of individuals $\vec{u}$ is allowed to be $1/2$. The set of all 3-valued structures over vocabulary $\mathcal{P}$ is denoted by $\mathcal{S}_3[\mathcal{P}]$. (We drop "$[\mathcal{P}]$" when $\mathcal{P}$ is clear from the context.)

Definition 1. The truth values 0 and 1 are *definite values*; $1/2$ is an *indefinite value*. For $l_1, l_2 \in \{0, 1/2, 1\}$, the *information order* is defined as follows: $l_1 \sqsubseteq l_2$ iff $l_1 = l_2$ or $l_2 = 1/2$. The symbol $\sqcup$ denotes the least-upper-bound operation with respect to $\sqsubseteq$.

The abstract stores used for program analysis are 3-valued logical structures that, by the construction discussed below, are *a priori* of bounded size. In general, each 3-valued logical structure corresponds to a (possibly infinite) set of 2-valued logical structures. Members of these two families of structures are related by *canonical abstraction*.

The principle behind canonical abstraction is illustrated in Fig. 4, which shows how 2-valued structure S is abstracted to 3-valued structure T. The abstraction function is determined by a subset $\mathcal{A}$ of the unary predicates. The predicates in $\mathcal{A}$ are called the *abstraction predicates*. Given $\mathcal{A}$, the act of applying the corresponding abstraction function is called $\mathcal{A}$*-abstraction*. The canonical abstraction illustrated in Fig. 4 is $\{x, y, z\}$-abstraction.

Abstraction is driven by the values of the "vector" of abstraction predicates for each individual w—i.e., for S, by the values $x(w)$, $y(w)$, and $z(w)$, for $w \in \{u_1, u_2, u_3, u_4\}$—and, in particular, by the equivalence classes formed from the individuals that have the same vector of values for their abstraction predicates. In S, there are two such equivalence classes: (i) $\{u_1\}$, for which x, y, and z are 1, 1, and 0, respectively, and (ii) $\{u_2, u_3, u_4\}$, for which x, y, and z are all 0. (The boxes in the table of unary predicates for S show how individuals of S are grouped into two equivalence classes.) All of the members of each equivalence class are mapped to the same individual of the 3-valued structure. Thus, all members of $\{u_2, u_3, u_4\}$ from S are mapped to the same individual in T, called u';[6] similarly, all members of $\{u_1\}$ from S are mapped to the same individual in T, called u.

For each non-abstraction predicate p^S of 2-valued structure S, the corresponding predicate p^T in 3-valued structure T is formed by a "truth-blurring quotient". The value for a tuple $\vec{u}_0$ in p^T is the join ($\sqcup$) of all p^S tuples that the equivalence relation on individuals maps to $\vec{u}_0$. For instance,

- In S, $n^S(u_1, u_1)$ equals 0. Therefore, the value of $n^T(u, u)$ is 0.
- In S, $n^S(u_2, u_1)$, $n^S(u_3, u_1)$, and $n^S(u_4, u_1)$ all equal 0. Therefore, the value of $n^T(u', u)$ is 0.
- In S, $n^S(u_1, u_3)$ and $n^S(u_1, u_4)$ both equal 0, whereas $n^S(u_1, u_2)$ equals 1; therefore, the value of $n^T(u, u')$ is $1/2$ $(= 0 \sqcup 1)$.
- In S, $n^S(u_2, u_3)$ and $n^S(u_3, u_4)$ both equal 1, whereas $n^S(u_2, u_2)$, $n^S(u_2, u_4)$, $n^S(u_3, u_2)$, $n^S(u_3, u_3)$, $n^S(u_4, u_2)$, $n^S(u_4, u_3)$, and $n^S(u_4, u_4)$ all equal 0; therefore, the value of $n^T(u', u')$ is $1/2$ $(= 0 \sqcup 1)$.

In Fig. 4, the boxes in the tables for predicates n and eq indicate these four groupings.

In a 2-valued structure, the eq predicate represents the equality relation on individuals. In general, under canonical abstraction some individuals "lose their identity" because of uncertainty that arises in the eq predicate. For instance, $eq^T(u, u) = 1$ because u in T represents a single individual of S. On the other hand, u' represents three individuals of S and the quotient operation causes $eq^T(u', u')$ to have the value $1/2$. An individual like u' is called a *summary individual*.

A 3-valued logical structure T is used as an abstract descriptor of a set of 2-valued logical structures. In general, a summary individual models a *set* of individuals in each

[6] The names of individuals are completely arbitrary: what distinguishes u' is the value of its vector of abstraction predicates.

of the 2-valued logical structures that T represents. The graphical notation for 3-valued logical structures (cf. structure T of Fig. 4) is derived from the one for 2-valued structures, with the following additions:

- Individuals are represented by circles containing their names. (In Fig. 5, discussed in §5, we also place non-0-valued unary predicates that do not correspond to pointer-valued program variables inside the circles.)
- A summary individual is represented by a double circle.
- Unary and binary predicates with value $1/2$ are represented by dotted arrows.

Thus, in every concrete structure $\tilde{S}$ that is represented by abstract structure T of Fig. 4, pointer variables `x` and `y` definitely point to the concrete node of $\tilde{S}$ that u represents. The `n`-field of that node may point to one of the concrete nodes that u' represents; u' is a summary individual, i.e., it may represent more than one concrete node in $\tilde{S}$. Possibly there is an `n`-field in one or more of these concrete nodes that points to another of the concrete nodes that u' represents, but there cannot be an `n`-field in any of these concrete nodes that points to the concrete node that u represents.

Note that 3-valued structure T also represents

- the acyclic lists of length 3 or more that are pointed to by `x` and `y`.
- the cyclic lists of length 3 or more that are pointed to by `x` and `y`, such that the backpointer is not to the head of the list, but to the second, third, or later element.
- some additional memory configurations with a cyclic or acyclic list pointed to by `x` and `y` that also contain some garbage cells that are not reachable from `x` and `y`.

That is, T is a finite representation of an infinite set of (possibly cyclic) concrete lists, each of which may also be accompanied by some unreachable cells. Later in this section, we discuss options for fine-tuning an abstraction. For instance, it is possible to use canonical abstraction to define abstractions in which the acyclic lists and the cyclic lists are mapped to different 3-valued structures (and in which the presence or absence of unreachable cells is readily apparent).

Canonical abstraction ensures that each 3-valued structure has an *a priori* bounded size, which guarantees that a fixed-point will always be reached by an iterative static-analysis algorithm. Another advantage of using 2- and 3-valued logic as the basis for static analysis is that the language used for extracting information from the concrete world and the abstract world is identical: *every* syntactic expression—i.e., every logical formula—can be interpreted either in the 2-valued world or the 3-valued world.[7]

The consistency of the 2-valued and 3-valued viewpoints is ensured by a basic theorem that relates the two logics, which eliminates the need for the user to write the usual proofs required with abstract interpretation—i.e., to demonstrate that the abstract descriptors that the analyzer manipulates correctly model the actual heap-allocated data structures that the program manipulates. Thanks to a single meta-theorem (the Embedding Theorem [1, Theorem 4.9]), which shows that information extracted from a 3-valued structure T by evaluating a formula φ is sound with respect to the value of φ in each of the 2-valued structures that T represents, an abstract semantics falls out automatically from a specification of the concrete semantics (which has to be provided

[7] Formulas are first-order formulas with transitive closure: a *formula* over the vocabulary $\mathcal{P} = \{eq, p_1, \ldots, p_n\}$ is defined as follows (where $p^*(v_1, v_2)$ stands for the reflexive transitive closure of $p(v_1, v_2)$):

$p \in \mathcal{P}, \varphi \in$ *Formulas*, $v \in$ *Variables*

$$\varphi ::= \mathbf{0} \mid \mathbf{1} \mid p(v_1, \ldots, v_k) \mid (\neg\varphi_1) \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 \vee \varphi_2) \mid (\exists v\colon \varphi_1) \mid (\forall v\colon \varphi_1) \mid p^*(v_1, v_2)$$

in any case whenever abstract interpretation is employed). In particular, the formulas that define the concrete semantics when interpreted in 2-valued logic define a sound abstract semantics when interpreted in 3-valued logic. Soundness of *all* instantiations of the analysis framework is ensured by the Embedding Theorem.

Instrumentation predicates. Unfortunately, unless some care is taken in the design of an analysis, there is a danger that as abstract interpretation proceeds, the indefinite value $1/2$ will become pervasive. This can destroy the ability to recover interesting information from the 3-valued structures collected (although soundness is maintained). A key role in combating indefiniteness is played by *instrumentation predicates*, which record auxiliary information in a logical structure. They provide a mechanism for the user to fine-tune an abstraction: an instrumentation predicate p of arity k, which is defined by a logical formula $\psi_p(v_1, \ldots, v_k)$ over the core predicate symbols, captures a property that each k-tuple of nodes may or may not possess. In general, adding additional instrumentation predicates refines the abstraction, defining a more precise analysis that is prepared to track finer distinctions among stores. This allows more properties of the program's stores to be identified during analysis.

p	**IntendedMeaning**	ψ_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?	$\exists\, v_1 \colon q(v_1) \wedge t[n](v_1, v)$
$c[n](v)$	Is v on a directed cycle of n-fields?	$\exists\, v_1 \colon n(v, v_1) \wedge t[n](v_1, v)$

Table 2. Defining formulas of some commonly used instrumentation predicates. Typically, there is a separate predicate symbol $r[n, q]$ for every pointer-valued variable q.

The introduction of unary instrumentation predicates that are then used 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. Instrumentation predicates that involve reachability properties, which can be defined using transitive closure, often play a crucial role in the definitions of abstractions. For instance, in program-analysis applications, reachability 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. Tab. 2 lists some instrumentation predicates that are important for the analysis of programs that use type `List`.

From the standpoint of the concrete semantics, instrumentation predicates represent cached information that could always be recomputed by reevaluating the instrumentation predicate's defining formula in the current store. From the standpoint of the abstract semantics, however, reevaluating a formula in the current (3-valued) store can lead to a drastic loss of precision. To gain maximum benefit from instrumentation predicates, an abstract-interpretation algorithm must obtain their values in some other way. This problem, the *instrumentation-predicate-maintenance problem*, is solved by incremental computation; the new value that instrumentation predicate p should have after a transition via abstract state transformer τ from state σ to σ' is computed incrementally from the known value of p in σ. An algorithm that uses τ and p's defining formula $\psi_p(v_1, \ldots, v_k)$ to generate an appropriate incremental predicate-maintenance formula for p is presented in [2].

The problem of automatically identifying appropriate instrumentation predicates, using a process of abstraction refinement, is addressed in [15]. 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 [2], provides a framework for eliminating previously required user inputs for which TVLA has been criticized in the past. Although the abstraction-refinement mechanism was not available for the experiments reported on in the present paper, we believe that it will work equally well when applied to the analysis of programs with recursive procedure calls. In particular, we have observed that the abstraction-refinement mechanism is capable of generating instrumentation predicates that record in/out relationships: most of the experiments described in [15] involved 2-vocabulary structures similar to those used in the present paper, and several of the instrumentation predicates identified relate pairs of predicates $p[in]/p[out]$.

Other operations on logical structures. Thanks to the fact that the Embedding Theorem applies to any pair of structures for which one can be embedded into the other, most operations on 3-valued structures need not be constrained to manipulate 3-valued structures that are images of canonical abstraction. Thus, it is not necessary to perform canonical abstraction after the application of each abstract structure transformer. To ensure that abstract interpretation terminates, it is only necessary that canonical abstraction be applied as a widening operator somewhere in each loop, e.g., at the target of each backedge in the CFG.

Several additional operations on logical structures help prevent an analysis from losing precision:

- 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 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 logical 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).

4 The Use of Logical Structures for Interprocedural Analysis

Given an abstract value A_0 that represents a set of initial stores, the goal is to compute—for each control point of each procedure—an overapproximation to the set of values for the local variables and the heap that can arise at that point. More precisely, the goal is to compute the "join-over-valid-paths" value for each node n:

$$\text{JOVP}(n) = \bigsqcup_{q \in \text{ValidPaths}(s_{main}, n)} \text{pf}_q(A_0)$$

where $\text{ValidPaths}(s_{main}, n)$ denotes the set of paths from s_{main} to n in which the call-to-start and exit-to-return-site edges in path q form a string in which each exit-to-return-site edge is balanced by a preceding call-to-start edge, and pf_q is the composition, in order, of the dataflow transformers for the edges of q.

Let $Id|_D$ denote the identity transformer restricted to inputs in D. For dataflow transformers that distribute over $\sqcup$, the JOVP solution can be obtained by finding the least solution to the following set of equations:

$$\phi(s_{main}) = Id|_{A_0} \quad A_0 \text{ describes the set of initial stores at } s_{main} \tag{1}$$

$$\phi(s_p) = Id|_D \quad s_p \in \text{StartNodes}, p \neq main, \text{ and } D = \bigsqcup_{(c, s_p) \in \text{CallToStartEdges}} \text{range}(\phi(c)) \tag{2}$$

$$\phi(n) = \bigsqcup_{(m,n)\in E^*} \tau_{m,n} \circ \phi(m) \qquad \text{for } n \in N^*,\ n \notin (\text{ReturnSites} \cup \text{StartNodes}) \tag{3}$$

$$\phi(n) = \phi(e_q) \circ \phi(\mathit{call}(n)) \qquad \text{for } n \in \text{ReturnSites, and } \mathit{call}(n) \text{ calls } q \tag{4}$$

Eqns. (1)–(4) can be understood as a variant of the "functional approach" of Sharir and Pnueli [5]; in [5], this is expressed with two fixed-point-finding phases: the first phase propagates transformer-valued values; the second phase propagates dataflow values proper. Eqns. (1)–(4) combine these into a single phase that propagates transformer-valued values only. Each summary transformer $\phi(n)$ is a partial function: the domain of $\phi(n)$ overapproximates the set of reachable states at $s_{\mathit{proc}(n)}$ from which it is possible to reach n; the range of $\phi(n)$ equals JOVP(n), which overapproximates the set of reachable states at n. (A two-phase approach à la Sharir and Pnueli [5] could also be used.[8])

To simplify the presentation, in §4.1 we will assume that the language does not support either local variables or parameter passing. In §4.2, we extend the approach to handle local variables and parameters.

4.1 A Simplified Setting: No Local Variables or Parameters

To use Eqns. (1)–(4) for interprocedural shape analysis, we follow Observation 1 and represent each $\phi(n)$ transformer as a set of 2-vocabulary 3-valued structures. As described below, suitable operations on 3-valued structures provide a way to compose such transformers.

The composition operation $\phi(e_q) \circ \phi(\mathit{call}(n))$ in Eqn. (4), which represents an interprocedural-propagation step, involves transformers represented by two sets of 2-vocabulary 3-valued structures. Intuitively, this involves collecting up a set of structures, where each structure is the "natural join" of two structures—one from each argument set. Below, we define the operation $T_2 \circ T_1$ for a single pair, T_2 and T_1.

In fact, to do this really requires three vocabularies: for each original predicate p, we use three predicates p[*in*], p[*out*], and p[*tmp*]. A 2-vocabulary 3-valued structure uses only p[*in*] and p[*out*]—or rather, the values of the p[*tmp*] predicates are "irrelevant". (When a predicate p is irrelevant, then $p(\vec{u})$ evaluates to $1/2$ for every tuple of individuals $\vec{u}$.) Another obstacle is to reconcile the values of the predicates in the different 2-vocabulary 3-valued structures. The solution has several parts:

- We need an operation to move predicates in one vocabulary to predicates in another vocabulary. The notation $T[\mathit{tmp} \leftarrow \mathit{out}; \mathit{out} \leftarrow 1/2]$ denotes the (simultaneous) transformation on structure T in which the p[*out*] predicates are moved to p[*tmp*], and the p[*out*] predicates are all set to $1/2$. For instance, to perform the composition $T_2 \circ T_1$, we use $T_1[\mathit{tmp} \leftarrow \mathit{out}; \mathit{out} \leftarrow 1/2]$ and $T_2[\mathit{tmp} \leftarrow \mathit{in}; \mathit{in} \leftarrow 1/2]$.
- We need structures that have the same sets of individuals. Because the individuals in 3-valued structures are identified by the values they have for the (unary) abstraction predicates, we use the operation $\mathit{canonical}\colon \mathcal{S}_3 \to \wp(\mathcal{S}_3)$, which refines a 3-valued structure T into a set of structures—each member of which is in the im-

[8] In the two-phase approach, the first phase is defined by Eqns. (1)–(4), except that the right-hand sides of Eqns. (1) and (2) are both replaced by *Id*. This permits summary transformers to be computed in a more modular fashion—i.e., bottom-up over the call graph's strongly-connected components. However, it also causes the analysis to consider more input possibilities for each procedure, which is an important consideration in our context. Eqns. (1)–(4) (as written) lead to a less modular analysis that requires a fixed-point iteration over the entire program.

age of canonical abstraction—such that the set describes the same set of concrete structures as T [16].

– We define the meet of two 3-valued structures that have the same set of individuals. Let $S_1 = (U, \iota_1)$ and $S_2 = (U, \iota_2)$ be two logical structures with the same universe U and vocabulary $\mathcal{P}$. The interpretations ι_1, ι_2 map each relation symbol $p \in \mathcal{P}_k$ to a k-ary truth-valued function: $\iota_i(p)\colon U^k \to \{0, 1/2, 1\}$. For convenience, we implicitly add a bottom element $\bot$ to the lattice $(\{0, 1, 1/2\}, \sqsubseteq)$ of Def. 1. The meet operator $S_1 \sqcap S_2$ is defined as

$$S_1 \sqcap S_2 \stackrel{\text{def}}{=} \begin{cases} (U, \iota_1 \sqcap \iota_2) & \text{if } \iota_1 \sqcap \iota_2 \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

where

$$\iota_1 \sqcap \iota_2 \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \iota_1(p)(\vec{u}) \sqcap \iota_2(p)(\vec{u}) = \bot \\ & \text{for some } p \in \mathcal{P}_k \text{ and } \vec{u} \in U^k \\ \lambda p \in \mathcal{P}_k . \lambda \vec{u} \in U^k . \iota_1(p)(\vec{u}) \sqcap \iota_2(p)(\vec{u}) & \text{otherwise} \end{cases}$$

If a predicate is irrelevant in S_1, its value in $S_1 \sqcap S_2$ is its value in S_2.

– We extend the previous definition to any pair of 3-valued structures by

$$S_1 \sqcap S_2 = \{S_1' \sqcap S_2' \mid S_1' \in \mathit{canonical}(S_1) \wedge S_2' \in \mathit{canonical}(S_2) \wedge U^{S_1'} = U^{S_2'}\} - \{\bot\} \tag{5}$$

With this notation, the composition of transformers $T_2 \circ T_1$, where T_1 and T_2 are 2-vocabulary 3-valued structures (which are really 3-vocabulary 3-valued structures) is expressed as follows:

$$T_2 \circ T_1 \stackrel{\text{def}}{=} \Big(T_1[\mathit{tmp} \leftarrow \mathit{out}; \mathit{out} \leftarrow 1/2] \sqcap T_2[\mathit{tmp} \leftarrow \mathit{in}; \mathit{in} \leftarrow 1/2]\Big)[\mathit{tmp} \leftarrow 1/2] \tag{6}$$

The effect is to perform a natural join on the $p[\mathit{tmp}]$ predicates to create structures that have T_1's $p[\mathit{in}]$ predicates, T_2's $p[\mathit{out}]$ predicates, and common $p[\mathit{tmp}]$ predicates. The $p[\mathit{tmp}]$ predicates are then eliminated by setting them to $1/2$.[9]

The composition operation is extended to sets of structures in the usual way:

$$SS_2 \circ SS_1 = \bigcup \{S_2 \circ S_1 \mid S_2 \in SS_2 \wedge S_1 \in SS_1\}.$$

In contrast, the composition operation $\tau_{m,n} \circ \phi(m)$ in Eqn. (3), which represents an intraprocedural-propagation step, is heterogeneous: $\tau_{m,n}$ is defined using a collection of *predicate-update formulas*, $c(v_1, \ldots, v_k) = \tau_{c,(m,n)}(v_1, \ldots, v_k)$, whereas $\phi(m)$ is a set of 2-vocabulary 3-valued structures. Thus, the composition operation in Eqn. (3) can be implemented merely by performing the standard TVLA intraprocedural-propagation step for $\tau_{m,n}$ on the *out* predicates (only) for each of the structures in $\phi(m)$. (Note that the $\sqcup$ operation in Eqn. (3) is union of sets of 3-valued structures. Each $\tau_{m,n}$ operates elementwise on a set of 3-valued structures, and hence distributes over $\sqcup$.)

In practice, Eqns. (1)–(4) are solved by propagating *changes* in values, rather than full values. Such a differential algorithm is presented in [17].

[9] A different view of this step is that making the $p[\mathit{tmp}]$ predicates irrelevant corresponds to existentially quantifying them out. If expressed by means of a formula, the operation of making $p[\mathit{tmp}]$ irrelevant would involve second-order quantification over $p[\mathit{tmp}]$; however, the operation is performed directly on a logical structure, and hence it is not a problem for us that the operation cannot be expressed by means of a first-order formula.

4.2 Local Variables and Parameters

Until now, we have assumed that a state of a program is defined by a memory configuration, and that relations between states are represented using structures over doubled vocabularies. Things are actually a bit more complicated: a state also includes the values of local variables, formal input parameters, and formal output parameters. The summary transformer $\phi(n)$ must thus also relate the values of formal input parameters at node $s_{proc(n)}$ to the state of the heap and the values of local variables at n.

To incorporate local variables and parameters, we merely have to expand the vocabulary to $\mathcal{P}_{loc} \uplus \mathcal{P}_g[in] \uplus \mathcal{P}_g[out] \uplus \mathcal{P}_g[tmp]$, where the vocabulary $\mathcal{P}_{loc}$ captures Boolean-valued and pointer-valued local variables and parameters, and $\mathcal{P}_g$ is the tripled vocabulary from §4.1. The assumption that formal input parameters are not modified in the body of a procedure makes it unnecessary to duplicate/triplicate the predicate symbols for parameters in $\mathcal{P}_{loc}$. Eqn. (2) then becomes:

$$\phi(s_q) = Id|_D \quad s_p \in \text{StartNodes}, p \neq main, \text{ and} \quad D = \bigsqcup_{\substack{(c,s_p)\in \text{CallToStartEdges} \\ \text{and the call is } \mathtt{y:=p(x)}}} \text{range}(\tau_{\mathsf{fpi}_p:=x} \circ \phi(c))[loc \setminus \{\mathsf{fpi}_p\} \leftarrow 1/2] \tag{7}$$

where $\tau_{:=}$ denotes the transformer generated by update formulas that correspond to the assignment in the subscript. Eqn. (7) reflects the binding of the actual parameter x at node c to the formal input parameter fpi_p at node s_p. All relations corresponding to the other local variables and parameters are set to irrelevant at this node.

For a call statement of the form y:=q(x), where $T_2 = \phi(e_q)$ and $T_1 = \phi(call(n))$, the transformer-composition operation $T_2 \circ T_1$ used in Eqn. (4) to implement the abstract procedure-return operation can be expressed as

$$T_2 \circ T_1 \stackrel{\text{def}}{=} \left(\tau_{y:=\mathsf{fpo}} \circ \left(\begin{array}{l} (\tau_{\mathsf{fpi}:=x} \circ T_1)[tmp \leftarrow out; out \leftarrow 1/2] \\ \sqcap \\ (\tau_{\substack{\mathsf{fpi}:=\mathsf{fpi}_q; \\ \mathsf{fpo}:=\mathsf{fpo}_q}} \circ T_2) \begin{bmatrix} tmp \leftarrow in; in \leftarrow 1/2; \\ loc \leftarrow 1/2 \end{bmatrix} \end{array} \right) \right) \begin{bmatrix} tmp \leftarrow 1/2; \\ \{\mathsf{fpi}, \mathsf{fpo}\} \leftarrow 1/2 \end{bmatrix} \tag{8}$$

where fpi and fpo are fresh unary core predicates (not in $\mathcal{P}_{loc}$ or $\mathcal{P}_g$) that are used to impose parameter-passing constraints as follows: fpi is bound to the value of the actual input parameter x of T_1; fpi is also bound to the value of formal input parameter fpi_q of T_2; and fpo is bound to the value of formal output parameter fpo_q of T_2. In particular, the fpi relation and all of the *tmp* relations are common in the meet operation performed in Eqn. (8). Then, because the local variables in T_2 are set to be irrelevant, the values for the local variables in the structures of the answer set are the values from T_1, with the exception of the actual output parameter y, which is assigned the value of $\mathsf{fpo} = \mathsf{fpo}_q$.

4.3 An Efficient Implementation of the Meet Operation

There are two sources of combinatorial explosion in Eqns. (4), (5), (6), and (8):

1. The number of pairs $(S_1, S_2) \in \phi(e_q) \times \phi(call(n))$ (quadratic explosion);
2. The cardinality of the sets $canonical(S_1)$ and $canonical(S_2)$ in Eqn. (5) defining the meet operator $\sqcap$ (exponential explosion).

Point 1 is inherited from the nature of our abstract lattice, which is a *powerset* domain, and the fact that we apply a binary operation (composition) to values in the domain. We do not address this problem here.

Point 2 is specific to our abstract lattice and concerns only the meet operation, especially when it is used to implement relational composition. Consider a pair of 3-valued structures T_1 and T_2 for which a composition is performed in Eqn. (4). In T_1, the core

predicates that represent variables of the called procedure q are irrelevant, so they have the value $1/2$. This means that the operation $canonical$ will enumerate all possible *definite* interpretations for these predicates; the number of these interpretations is exponential in the number of such predicates. A similar situation holds for T_2.

More generally, consider a structure $S = \langle U^S, \iota^S \rangle$ with n irrelevant unary core predicates; the cost of $canonical$ is $\mathcal{O}((2^{|U^S|})^n)$. Even if the unary predicates represent only pointer-valued variables, which means that such predicates may evaluate to 1 on at most one individual, there are still $\mathcal{O}(|U|^n)$ possible interpretations.

In our case, this combinatorial explosion is all the more frustrating because it is only temporary: the meet $S_1 \sqcap S_2$ will reject (by evaluating to $\bot$) most of the structures obtained by enumerating definite interpretations of irrelevant predicates in S_1 (resp. S_2). Indeed, predicates that are irrelevant in one structure and relevant in the other usually have definite interpretations in the latter.

A better implementation of $\sqcap$. The approach that we actually followed in our extended version of TVLA was to implement an approximation to the meet operation using systems of 3-valued constraints [1], which were already supported by the base TVLA system. In TVLA, there is a global set of constraints C_0 that is used to express integrity constraints on the set of 2-valued structures that a 3-valued structure represents. For instance, some of the constraints in C_0 express the fact that a unary predicate that represents a pointer-valued variable can evaluate to 1 on at most one individual. For convenience, we will associate a constraint set C^S with each structure, so that a 3-valued structure S is now a triple: $\langle U^S, \iota^S, C^S \rangle$. ($C^S$ is generally C_0.)

A set of constraints C represents the set of concrete structures that satisfy C:

$$\gamma_c(C) \stackrel{\text{def}}{=} \{S \in \mathcal{S}_2 \mid S \models C\} \tag{9}$$

in the same way that a 3-valued structure S represents the set of concrete structures $\gamma(\{S\})$ that can be embedded into S via canonical abstraction [1].

Assume now that we have an operation $cons : \mathcal{S}_3 \to \wp(\mathcal{C})$ that associates to a given structure a set of constraints such that for any S, $\gamma(\{S\}) \subseteq \gamma_c(cons(S))$. In other words, constraint set $cons(S)$ overapproximates S. For any logical structures $S_1 = \langle U^{S_1}, \iota^{S_1}, C^{S_1} \rangle$ and $S_2 = \langle U^{S_2}, \iota^{S_2}, C^{S_2} \rangle$, we now define the operation $\sqcap^c$:

$$S_1 \sqcap^c S_2 \stackrel{\text{def}}{=} \langle U^{S_1}, \iota^{S_1}, C^{S_1} \cup cons(S_2) \rangle.$$

This operator has the following property: $\gamma(\{S_1 \sqcap^c S_2\}) \supseteq \gamma(\{S_1\}) \cap \gamma(\{S_2\})$, with equality if the $cons$ operation is exact.

To summarize, the approximate meet operator consists of adding $cons(S_2)$ to S_1 temporarily, then performing Focus and Coerce operations to transfer the information that is initially contained in the additional constraints to the universe U^{S_1} and the interpretation ι^{S_1}. Afterwards, the additional constraints are removed.

For instance, when we use the meet operation in Eqns. (6) and (8), we replace $S'_1 \sqcap S'_2$ in Eqn. (5) by $Coerce(Focus(S'_1 \sqcap^c S'_2, \{\mathsf{fpo}(v)\}))$. This forces fpo to be given a definite interpretation that is constrained by the set $cons(S_2)$, which represents the summary transformer of the callee.

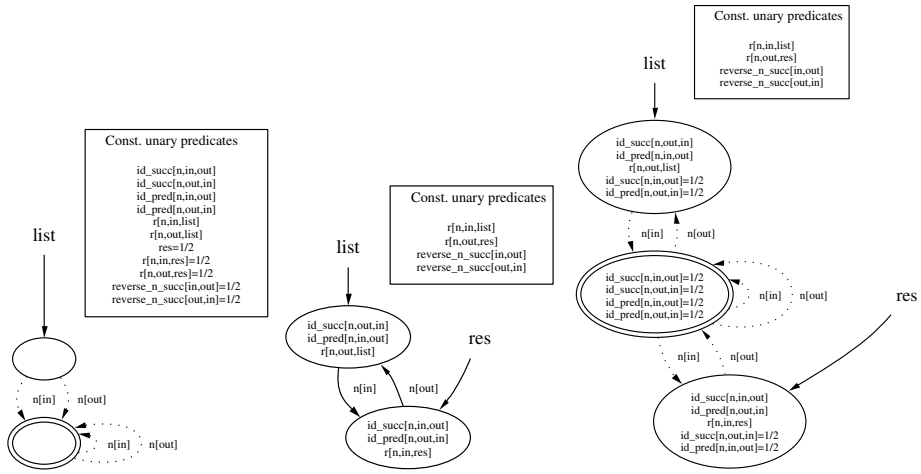
Converting a 3-valued structure to a set of constraints. To achieve this, we adapted a result from [18], which shows how to characterize a 3-valued logical structure that is in the image of canonical abstraction by means of a formula in first-order logic with transitive closure. The resulting formula can easily be converted to a set of constraints

that satisfy the restricted syntax given in [1]. However, one of the constraints that would be generated according to [18] would be too expensive to check from an algorithmic point of view, so this constraint is dropped, which induces a safe overapproximation. (Roughly, this constraint captures the fact that any *concrete* structure represented by the abstract structure should contain a number of individuals greater than or equal to the number of individuals in the abstract structure.)

5 Implementation and Experiments

To perform interprocedural shape analysis by the method that is described in §4, we created a modified version of TVLA [3], 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*.
- We designed a more general language in which to specify equation systems.
- We implemented an approximation to the meet operation on 3-valued structures (and hence to the composition operation), as described in §4.3.



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

Fig. 5. List-reversal example. (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.)

Fig. 5 shows how the summary information we obtain captures the behavior of the recursive list-reversal procedure of Figs. 1 and 2. The descriptor of the initial summary transformer at start node s_{main} was the 3-valued structure S_0, shown in Fig. 5(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 Fig. 5(b)–(c), which represent the transformations that reverse lists of length two, and all lists of length greater than two, respectively.

As discussed in §3, to prevent the loss of essential information, several families of instrumentation predicates were introduced:

- The unary predicates $id_succ[n, m_1, m_2]$ and $id_pred[n, m_1, m_2]$, where $m_1, m_2 \in \{in, out\}$ and $m_1 \neq m_2$, record information about the values of different modes

of predicate n, in particular, whether the value of predicate $n[m_1]$ implies $n[m_2]$. These are defined by

$$\textit{id_succ}[n, m_1, m_2](v) = \forall v_1 : (n[m_1](v, v_1) \Rightarrow n[m_2](v, v_1))$$
$$\textit{id_pred}[n, m_1, m_2](v) = \forall v_1 : (n[m_1](v_1, v) \Rightarrow n[m_2](v_1, v)).$$

The fact that $\textit{id_succ}[n, \textit{in}, \textit{out}](v) \wedge \textit{id_succ}[n, \textit{out}, \textit{in}](v) \wedge \textit{id_pred}[n, \textit{in}, \textit{out}](v) \wedge \textit{id_pred}[n, \textit{out}, \textit{in}](v)$ holds globally in S_0 (*cf.* Fig. 5(a)) captures the condition that the $n[\textit{in}]$ and $n[\textit{out}]$ predicates are identical at the entry node of the procedure. The $n[\textit{in}]$ predicates serve as an indelible record of the state of the n-links at the entry node.

– The unary predicates $\textit{reverse_n_succ}[m_1, m_2]$, again with $m_1, m_2 \in \{\textit{in}, \textit{out}\}$ and $m_1 \neq m_2$, record whether $n[m_2]$ is an inverse of $n[m_1]$. These are defined by

$$\textit{reverse_n_succ}[m_1, m_2](v) = \forall v_1 : (n[m_1](v, v_1) \Rightarrow n[m_2](v_1, v)).$$

The values for these predicates in S_1 and S_2 show that for each n-link $n[\textit{in}](v_1, v_2)$ at the entry node $s_{\textit{main}}$, we have an n-link $n[\textit{out}](v_2, v_1)$ at the exit node $e_{\textit{main}}$. In other words, the procedure has reversed all the n-links.

In addition, during the composition operation, some additional constraint rules were needed for the system to be able to deduce a relationship between $n[\textit{in}]$ and $n[\textit{out}]$. These are defined by

$$\textit{id_succ}[n, \textit{in}, \textit{tmp}](v) \wedge \textit{reverse_n_succ}[\textit{tmp}, \textit{out}](v) \Rightarrow \textit{reverse_n_succ}[\textit{in}, \textit{out}](v)$$
$$\textit{reverse_n_succ}[\textit{in}, \textit{tmp}](v) \wedge \textit{id_pred}[\textit{tmp}, \textit{out}](v) \Rightarrow \textit{reverse_n_succ}[\textit{in}, \textit{out}](v)$$

Notice that only the $\textit{reverse_n_succ}[m_1, m_2]$ predicates and the related constraint rules are particular to the list-reversal example. The other predicates that appear in Fig. 5 were already used in previous papers on shape analysis of list-manipulation programs (see [1]): for instance, $r[n, \textit{out}, \textit{list}](v)$ holds the value 1 for individuals that are reachable from variable `list` through a chain of $n[\textit{out}]$ links. From the above definitions of the instrumentation predicates, it should be clear that the set of 3-valued structures $\{S_1, S_2\}$ accurately captures the fact that the output list is the reversal of the input list, and that the result is a list of length at least two.

Our second experiment involved comparing our results with [11] on the following examples: (i) list reversal (as discussed above), and (ii) non-deterministic insertion and deletion of a cell in a list. Results are shown in Fig. 6. Our method performs better than that of [11] for the list-reversal program, but worse for the latter two programs. For those, we considered programs where the cell to be inserted is passed as an input parameter (in the insert example), and the deleted cell is received back as an output parameter (in the delete example); this provides information about where the cell has been inserted (resp. deleted). For the versions of the programs analyzed by the method of [11], we added a global variable `cell`, which plays a similar role.

Program	Our method			Method of [11]		
	# of structs.	Time (sec)	Space (Mb)	# of structs.	Time (sec)	Space (Mb)
reverse	7/3	11	26	· /3	37	17
insert	23/9	188	43	· /9	70	18
delete	32/13	222	43	· /7	47	17
tree exch.	22/10	92	33	—		

The experiments were performed on a PC equipped with a 2 GHz Pentium 4 processor and 768 Mb of memory. *Time* and *Space* information were obtained with the `time` and `top` commands. The two numbers in each entry of the columns labeled *# of structs.* give the number of structures for the summary transformer of the recursive procedure and the number of structures at the end of the main procedure, respectively.

Fig. 6. Experimental results

Concerning the slower computation times, we think they are mainly due to the higher number of predicates to be manipulated (because of the different modes) and the cost of the meet operation. However, it is important to keep in mind that our method computes a *summary transformer* for each procedure, which [11] does not. The summary transformer $\phi(e_q)$ at an exit node e_q is a partial function: the domain of $\phi(e_q)$ overapproximates the set of reachable states at $s_{proc(e_q)}$ from which it is possible to reach e_q; the range of $\phi(e_q)$ overapproximates the set of reachable states at e_q. This has an impact on the results: for the delete example, the method of [11] is not able to keep track of the original position in the list of the deleted cell, unlike our method. (For the insert example, however, the two methods are similar w.r.t. this kind of information.)

Our third experiment was to analyze a procedure that recursively exchanges the right and left subtrees of a binary tree. This example is interesting because it would be difficult to implement this operation as a non-recursive procedure. The analysis was able to establish that after the procedure finishes, the subtrees of all cells reachable from the root have been exchanged, whereas the other cells have not been modified.

Statistics are given in Fig. 6. More information about the experiments is available at http://www.irisa.fr/prive/bjeannet/interproctvla/interproctvla.html.

6 Related Work

The analysis described in this paper 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 [19], octagons [20], etc.). In contrast, Observation 1 provides a way to create composable abstract transformers for the analysis of programs that support both dynamically-allocated storage and destructive updating of pointer-valued fields of structures. A key feature of the approach is that instrumentation predicates can refer to both the $\mathcal{P}[in]$ and $\mathcal{P}[out]$ vocabularies. For instance, the family of unary predicates $reverse_n_succ[m_1, m_2]$ discussd in §5 (with $m_1, m_2 \in \{in, out\}$ and $m_1 \neq m_2$) records whether $n[m_2]$ is an inverse of $n[m_1]$.

As discussed in the introduction, interprocedural shape analysis was also studied in [11]. The approach used in the present paper was inspired by the functional approaches of [4–6]. In contrast, the approach used in [11] is more reminiscent of the "call-strings" approach of [5].

A method for performing interprocedural shape analysis using procedure specifications and assume-guarantee reasoning is presented in [16]. There it is assumed that a specification for each procedure—a pre- and post-condition—is already known; the technique presented in [16] can be used to interpret a procedure's pre- and post-condition 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 post-condition holds. Similarly, when a procedure is analyzed, the pre-condition is assumed to hold on entry, and at end of the procedure the post-condition is checked. The work described in the present paper is *complementary* to [16]: the work described here—particularly in the modified form sketched in footnote 8—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 [16].

A second connection is that [16] provides a method to compute the most-precise overapproximation of the meet of two abstract values, which is the operation needed for composing transformers that are expressed as sets of 2-vocabulary 3-valued structures (see Eqns. (6) and (8)). Consequently, [16] provides a more precise alternative to the approximate meet operation described in §4.3. (At present, implementations of the methods from [16] are based on theorem provers, and are much slower than the method from §4.3, which does not involve a theorem prover.)

Acknowledgments. We thank Viktor Kuncak for several helpful discussions about the approach taken in this paper.

References

1. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. Trans. on Prog. Lang. and Syst. **24** (2002) 217–298
2. Reps, T., Sagiv, M., Loginov, A.: Finite differencing of logical formulas for static analysis. In: European Symp. on Programming. (2003) 380–398
3. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Static Analysis Symp. (2000) 280–301
4. Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In Neuhold, E., ed.: Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977). North-Holland (1978) 237–277
5. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In Muchnick, S., Jones, N., eds.: Program Flow Analysis: Theory and Applications. Prentice-Hall, Englewood Cliffs, NJ (1981) 189–234
6. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: Comp. Construct. (1992) 125–140
7. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Princ. of Prog. Lang., New York, NY, ACM Press (1995) 49–61
8. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comp. Sci. **167** (1996) 131–170
9. Reps, T., Schwoon, S., Jha, S.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: Static Analysis Symp. (2003) 189–213
10. Ball, T., Rajamani, S.: Bebop: A path-sensitive interprocedural dataflow engine. In: Prog. Analysis for Softw. Tools and Eng. (2001) 97–103
11. Rinetzky, N., Sagiv, M.: Interprocedural shape analysis for recursive programs. In: Comp. Construct. Volume 2027 of Lec. Notes in Comp. Sci. (2001) 133–149
12. Jones, N., Muchnick, S.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: Princ. of Prog. Lang. (1982) 66–74
13. Deutsch, A.: On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In: Princ. of Prog. Lang. (1990) 157–168
14. Gopan, D., DiMaio, F., N.Dor, Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Tools and Algs. for the Construct. and Anal. of Syst. (2004) 512–529
15. Loginov, A., Reps, T., Sagiv, M.: Abstraction refinement for 3-valued-logic analysis. Tech. Rep. 1504, Comp. Sci. Dept., Univ. of Wisconsin (2004)
16. Yorsh, G., Reps, T., Sagiv, M.: Symbolically computing most-precise abstract operations for shape analysis. In: Tools and Algs. for the Construct. and Anal. of Syst. (2004) 530–545
17. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. Tech. Rep. 1505, Comp. Sci. Dept., Univ. of Wisconsin (2004)
18. Yorsh, G.: Logical characterizations of heap abstractions. Master's thesis, School of Computer Science, Tel Aviv University, Israel (2003)
19. Cousot, P., Halbwachs, N.: Automatic discovery of linear constraints among variables of a program. In: Princ. of Prog. Lang. (1978) 84–96
20. Miné, A.: The octagon abstract domain. In: 8th Working Conf. on Rev. Eng. (2001) 310–322