Refinement-Based Verification for Possibly-Cyclic Lists*

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

[1] IBM T.J. Watson Research Center; alexey@us.ibm.com
[2] Comp. Sci. Dept., University of Wisconsin; reps@cs.wisc.edu
[3] School of Comp. Sci., Tel-Aviv University; msagiv@post.tau.ac.il

Abstract. In earlier work, we presented an abstraction-refinement mechanism that was successful in verifying automatically the partial correctness of *in-situ* list reversal when applied to an acyclic linked list [10]. This paper reports on the automatic verification of the *total correctness* (partial correctness and termination) of the same list-reversal algorithm, when applied to a *possibly-cyclic* linked list. A key contribution that made this result possible is an extension of the *finite-differencing* technique [14] to enable the maintenance of reachability information for a restricted class of possibly-cyclic data structures, which includes possibly-cyclic linked lists.

1 Introduction

Reinhard Wilhelm has long been associated with the Dagstuhl Seminars on Computer Science. In March of 2003, the Dagstuhl Seminar "Reasoning about Shape" was dedicated to one of the subjects that benefited from important contributions on the part of Reinhard Wilhelm. During that seminar, Richard Bornat posed an interesting challenge problem to the authors. The challenge concerns the application of the *in-situ* list reversal procedure `Reverse` to a *panhandle list*, i.e., a linked list that contains a cycle but in which at least the head of the list is not part of the cycle. (The lists shown in Fig. 1 are examples of panhandle lists.) Richard Bornat challenged us to use our techniques to demonstrate that, when applied to a panhandle list, `Reverse` produces a list in which the orientation of the successor edges in the panhandle (the acyclic part of the list) is as it was in the input list, while the orientation of the successor edges on the cycle is reversed.

In [10], we presented an abstraction-refinement mechanism for use in static analyses based on 3-valued logic [17], where the semantics of statements and the query of interest are expressed using logical formulas. Our abstraction-refinement mechanism introduces additional *instrumentation relations* (defined via logical formulas over *core relations*, which capture the basic properties of memory configurations). Instrumentation relations record auxiliary information in a logical structure, thus providing a mechanism to fine-tune an abstraction: an instrumentation relation captures a property that an individual memory cell may or may not possess. In general, the introduction of additional instrumentation relations refines an abstraction into one that is prepared to track

* Supported by ONR (N00014-01-1-{0708,0796}) and NSF (CCR-9986308 and CCF-{0524051,0540955}).

** The work was performed while Loginov was at the University of Wisconsin.

finer distinctions among stores. This allows more properties of the program's stores to be identified. The abstraction-refinement mechanism made possible the automatic verification of a number of interesting properties, including the partial correctness of *in-situ* list reversal when applied to an *acyclic* linked list.

In our context, the semantics of statements is expressed using logical formulas that describe changes to core-relation values. When instrumentation relations have been introduced to refine an abstraction, the challenge is to reflect the changes in core-relation values in the values of the instrumentation relations. To address this challenge, the authors presented *finite differencing*, a technique that constructs automatically *instrumentation-relation maintenance formulas*, the part of abstract transformers that deals with instrumentation relations [14].

A key aspect of the finite-differencing technique is its handling of reachability instrumentation relations, i.e., relations defined via the transitive-closure operator. In [14], we adapted a result by Dong and Su [2] to enable the maintenance of reachability information for acyclic data structures purely in first-order logic, i.e., without the recomputation of transitive closure, which generally results in a loss of precision.

In this paper, we reduce the problem of reachability maintenance for possibly-cyclic lists, e.g., panhandle lists, to the problem of reachability maintenance in acyclic data structures. The essential problem is that all nodes in the cyclic part of a panhandle list "look the same" in some sense, and the key to a solution is finding a way to break the symmetry of the cycle. (This is discussed further in §3.) The key idea—inspired by a similar idea used by William Hesse in his Ph.D. thesis—is to "break" each cycle: we define a binary instrumentation relation sfe_n to include all edges of the data structure, except one designated edge on each cycle. We define an additional instrumentation relation, sfp_n, to be the reflexive transitive closure of the acyclic relation sfe_n.[4] The relation sfp_n can be maintained using our prior results for acyclic reachability maintenance. Reachability information in the actual (possibly-cyclic) data structure can then be computed based on sfp_n.

This reduction addresses the shortcoming of finite differencing that prevented our techniques from establishing interesting properties of programs that manipulate possibly-cyclic linked lists. We show that, equipped with the extended finite-differencing technique, the abstraction-refinement mechanism is capable of introducing instrumentation relations that are sufficient to encode the key properties of `Reverse` when applied to possibly-cyclic linked lists. The contributions of this paper can be summarized as follows:

- We present an extension of finite differencing that allows first-order-logic maintenance of reachability information in possibly-cyclic linked lists. This is achieved via a reduction to the problem of reachability maintenance in acyclic data structures.
- We demonstrate the use of a *Data-Structure Constructor* for constructing an abstract representation of all possibly-cyclic linked lists, including panhandle lists.

[4] As discussed later, sfe_n and sfp_n stand for "spanning-forest edge" and "spanning-forest path", respectively.

– We demonstrate the use of automatic abstraction refinement for introducing the instrumentation relations that are sufficient for verifying the partial correctness of `Reverse` when applied to any possibly-cyclic linked list.
– We present a simple progress monitor that allows the analysis to establish the termination of `Reverse` on any possibly-cyclic linked list.

The contributions fall into two categories: (i) extending the scope of finite differencing so that reachability information can be maintained for possibly-cyclic lists, and (ii) the application of abstraction refinement for verifying properties of `Reverse`. The former contribution category is discussed in §3. The latter contribution category is discussed in §6.

An advantage of our abstract-interpretation approach is that it does not require the use of a theorem prover. This is particularly beneficial in our setting because our logic is undecidable [5].

2 Program Analysis using 3-Valued Logic

In this section, we give a brief overview of the framework of parametric shape analysis via 3-valued logic. For more details, the reader is referred to [17].

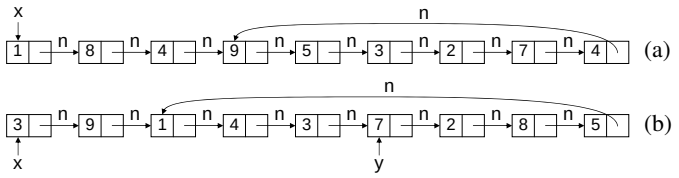

Fig. 1. Possible stores for *panhandle* linked lists. (a) A panhandle list pointed to by `x`. We will refer to lists of this shape as type-X lists. (b) A panhandle list pointed to by `x` with `y` pointing into the middle of the cycle. We will refer to lists of this shape as type-XY lists.

Program states are represented using *first-order logical structures*, which consist of a collection of *individuals*, together with an *interpretation* for a finite vocabulary of finite-arity relation symbols, $\mathcal{R}$. An interpretation is a truth-value assignment for each relation symbol for every appropriate-arity tuple of individuals. To ensure termination, the framework puts a bound on the number of distinct logical structures that can arise during analysis by grouping individuals that are indistinguishable according to a special subset of unary relations, $\mathcal{A}$. The grouping of nodes is referred to as *canonical abstraction* and the set $\mathcal{A}$ is referred to as the set of *abstraction relations*.

The application of canonical abstraction typically transforms a logical structure S into a *3-valued logical structure* $S^{\#}$, in which the third value, $1/2$, denotes the possibility of having either 0 (false) or 1 (true) in S. A program state is updated and queried via logical formulas, which are interpreted over the 3-valued structure $S^{\#}$ using a straightforward extension of Kleene's 2-valued semantics.

Because of canonical abstraction, an individual in a 3-valued structure can represent more than one individual in a given 2-valued structure; such an individual is referred to as a *summary individual*. In general, a 3-valued logical structure can represent an infinite set of 2-valued structures.

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

(a)

Relation	Intended Meaning
$x(v)$	Does pointer variable x point to memory cell v?
$n(v_1, v_2)$	Does the n field of v_1 point to v_2?

(b)

Table 1. (a) Declaration of a linked-list datatype in C; (b) core relations used for representing the stores manipulated by programs that use type `List`.

Program states are encoded in terms of *core relations*, $\mathcal{C} \subseteq \mathcal{R}$. Core relations are part of the underlying semantics of the language to be analyzed; they record atomic properties of stores. For instance, Tab. 1 gives the definition of a C linked-list datatype, and lists the core relations that would be used to represent the stores manipulated by programs that use type `List`, such as the stores in Fig. 1. Unary relations represent pointer variables, and binary relation n represents the n field of a `List` cell. Fig. 2(a) shows 2-valued structure S_2, which represents the store of Fig. 1(a) using the relations of Tab. 1.

p	Intended Meaning	Defining Formula
$is_n(v)$	Do n fields of two or more list nodes point to v?	$\exists\, v_1, v_2 \colon n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$
$r_{n,x}(v)$	Is v reachable from pointer variable x along n fields?	$\exists\, v_1 \colon x(v_1) \wedge n^*(v_1, v)$
$c_n(v)$	Is v on a directed cycle of n fields?	$n^+(v, v)$

Table 2. Defining formulas of instrumentation relations commonly employed in analyses of programs that use type `List`. The relation name is_n abbreviates "is-shared". There is a separate **r**eachability relation $r_{n,x}$ for every program variable x.

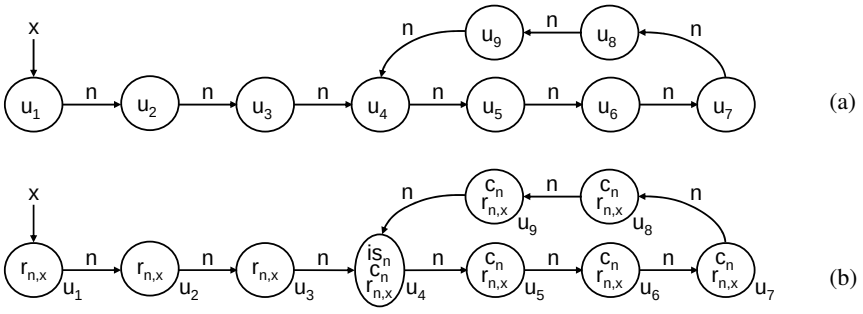

Fig. 2. A logical structure S_2 that represents the store shown in Fig. 1(a) in graphical form: (a) S_2 with relations of Tab. 1; (b) S_2 with relations of Tabs. 1 and 2.

The abstraction function on which an analysis is based, and hence the precision of the analysis defined, can be tuned by (i) choosing to equip structures with additional *instrumentation relations* to record derived properties, and (ii) varying which of the unary core and unary instrumentation relations are used as the set of abstraction relations. The set of instrumentation relations is denoted by $\mathcal{I}$. Each arity-k relation symbol is defined by an *instrumentation-relation defining formula* with k free variables. Instrumentation-relation symbols may appear in the defining formulas of other instrumentation relations as long as there are no circular dependences.

Tab. 2 lists some instrumentation relations that are important for the analysis of programs that use type `List`. Instrumentation relations that involve reachability properties, such as relation $r_{n,x}(v)$, often play a crucial role in the definitions of abstractions. These relations have the effect of keeping disjoint sublists summarized separately. Fig. 2(b) shows 2-valued structure S_2, which represents the store of Fig. 1(a) using the core relations of Tab. 1, as well as the instrumentation relations of Tab. 2.

If all unary relations are abstraction relations, the canonical abstraction of 2-valued logical structure S_2 is S_3, shown in Fig. 3, with list nodes corresponding to u_2 and u_3 in S_2 represented by the summary individual u_2 of S_3 and list nodes corresponding to $u_5, \ldots, u_9$ in S_2 represented by the summary individual u_4 of S_3. S_3 represents any type-X panhandle list with at least two nodes in the panhandle and at least two nodes in the cycle. The following graphical notation is used for depicting 3-valued logical structures:

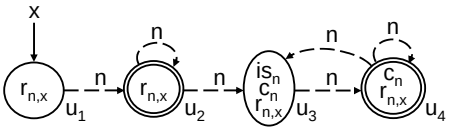

Fig. 3. A 3-valued structure S_3 that is the canonical abstraction of structure S_2. In addition to S_2, S_3 represents any type-X panhandle list with at least two nodes in the panhandle and at least two nodes in the cycle.

- Individuals are represented by circles containing (non-0) values for unary relations. Summary individuals are represented by double circles.
- A unary relation p corresponding to a pointer-valued program variable is represented by a solid arrow from p to the individual u for which $p(u) = 1$, and by the absence of a p-arrow to each node u' for which $p(u') = 0$. (If $p = 0$ for all individuals, the relation name p is not shown.)
- A binary relation 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 $q(u'_i, u'_j) = 0$.
- Relations with value $1/2$ are represented by dotted arrows.

For each kind of statement in the programming language, the concrete semantics is defined by *relation-update formulas* for core relations. The structure transformers for the abstract semantics are defined by the same relation-update formulas for core relations and *relation-maintenance formulas* for instrumentation relations. The latter are generated automatically via *finite differencing* [14]. Abstract interpretation collects a set of 3-valued structures at each program point. It is implemented as an iterative procedure that finds the least fixed point of a certain set of equations [17]. When the fixed point is

reached, the structures that have been collected at a program point describe a superset of all the execution states that can arise there.

Not all logical structures represent admissible stores. To exclude structures that do not, we impose integrity constraints. For instance, relation $x(v)$ of Tab. 1 captures whether pointer variable x points to memory cell v; x would be given the attribute "unique", which imposes the integrity constraint that x can hold for at most one individual in any structure: $\forall\, v_1, v_2 \colon x(v_1) \wedge x(v_2) \Rightarrow v_1 = v_2$. This formula evaluates to 1 in any 2-valued logical structure that corresponds to an admissible store. Integrity constraints contribute to the concretization function (γ) for our abstraction [18]. Integrity constraints are enforced by *coerce*, 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 an integrity constraint is definitely violated by the structure (e.g., if the structure cannot represent any admissible store).

3 Reachability Maintenance in Possibly-Cyclic Linked Lists

Unfortunately, the relations defined in Tabs. 1 and 2 do not permit precise maintenance of reachability information, such as relation $r_{n,x}$, in possibly-cyclic lists. A difficulty arises when reachability information has to be updated to reflect the deletion of an n edge on a cycle (e.g., as a result of statement `y->n = NULL`). With the relations defined in Tabs. 1 and 2, such an update requires the recomputation of a transitive-closure formula, which generally results in a drastic loss of precision in the presence of abstraction.

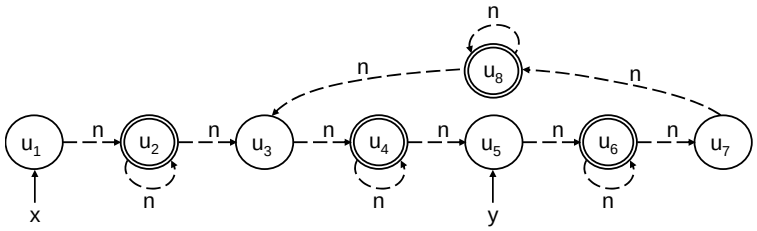

Fig. 4. Logical structure S_4 that represents type-XY panhandle lists, such as the store of Fig. 1(b). The relations of Tab. 2 are omitted to reduce clutter. Their values are as expected for a type-XY list: $r_{n,x}$ holds for all nodes, $r_{n,y}$ and c_n hold for all nodes on the cycle, and is_n holds for u_3.

We demonstrate the issue on panhandle lists represented by the abstract structure S_4 shown in Fig. 4, i.e., lists of type XY. Statement `y->n = NULL` has the effect of deleting the n edge leaving u_5, thus making the nodes represented by u_6, u_7, and u_8 unreachable from x.[5] Note that a first-order-logic formula over the relations of Tabs. 1 and 2 cannot distinguish the list nodes represented by u_4 from those represented by u_6, u_7, and u_8: all of those nodes are reachable from both x and y, none of those nodes are shared, and all of them lie on a cycle. Our inability to characterize

[5] Clearly, all nodes except u_5 also become unreachable from y.

the group of nodes represented by u_4 via a first-order formula requires the maintenance formula for the reachability relation $r_{n,x}$ to recompute some transitive-closure information, e.g., the transitive-closure subformula of the definition of $r_{n,x}$, namely, $n^*(v_1, v)$. However, in the presence of abstraction, recomputing transitive-closure formulas often yields $1/2$. For instance, in S_4, formula $n^*(v_1, v)$ evaluates to $1/2$ under the assignment $[v_1 \mapsto u_1, v \mapsto u_4]$ because of the many $1/2$ values of relation n (see the dashed edges connecting u_1 with u_2, for example).

The essence of a solution that enables maintaining reachability relations for possibly-cyclic lists in first-order logic is to find a way to break the symmetry of each cycle. The basic idea for a solution was suggested to us by William Hesse and Neil Immerman. It consists of maintaining a spanning-tree representation of a possibly-cyclic list. Reachability in such a representation can be maintained using first-order-logic formulas. Reachability in the actual list can be expressed in first-order logic based on the spanning-tree representation. We now explain our approach and highlight some differences with the approach taken by Hesse [4].

Our approach relies on the introduction of additional core and instrumentation relations. We extend the set of core relations (Tab. 1) with unary relation roc_n, which designates one node on each cycle to be the **r**epresentative **o**f the **c**ycle. (We refer to such a node as a roc_n node.) Relation roc_n is used for tracking a unique *cut edge* on each cycle, which allows the maintenance of a spanning tree. Fig. 5(a) shows 2-valued structure S_5, which represents the store of Fig. 1(a) using the extended set of core relations. Here, we let u_7 be the roc_n node. In general, we simply require that exactly one node on each cycle be designated as a roc_n node. Later in this section we describe how we ensure this.

Tab. 3 lists the extended set of instrumentation relations. Note that relation roc_n is not part of the semantics of the language. A natural question is whether $roc_n(v)$ can be defined as an instrumentation relation. For instance, we can try to define it using the following defining formula:

$$c_n(v) \wedge \exists\, v_1 \colon n(v_1, v) \wedge \neg c_n(v_1) \tag{1}$$

Formula (1) identifies nodes that lie on a cycle but have a predecessor that does not. There are three problems with this approach. First, this definition works for panhandle lists but not for cyclic lists without a panhandle. (In general, no other definition can work for cyclic lists without a panhandle because if one existed, it would need to choose one list node among identical-looking nodes that lie on each cycle.) Second, because the cyclicity relation c_n is defined in terms of roc_n (and sfp_n), the definition of roc_n has a circular dependence, which is disallowed. (This circularity cannot be avoided, if we want all reachability relations to benefit from the precise maintenance of one transitive-closure relation—here, sfp_n.) The third problem with introducing roc_n as an instrumentation relation is discussed later in the section (in footnote 6).

We divide our description of the abstraction based on the new set of relations into three parts, which describe (i) how the relations of Tab. 3 define *directed* spanning forests, (ii) how we maintain precision on a cycle in the presence of abstraction, and (iii) how we generate maintenance formulas for instrumentation relations *automatically*. The three parts highlight the differences between our approach and that of Hesse.

p	**Intended Meaning**	**Defining Formula**
$is_n(v)$	Do n fields of two or more list nodes point to v?	$\exists\, v_1, v_2 \colon n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$
$sfe_n(v_1, v_2)$	Is there an n edge from v_2 to v_1 (assuming that v_2 is not a roc_n node)	$n(v_2, v_1) \wedge \neg roc_n(v_2)$
$sfp_n(v_1, v_2)$	Is v_2 reachable from v_1 along sfe_n edges?	$sfe_n^*(v_1, v_2)$
$t_n(v_1, v_2)$	Is v_2 reachable from v_1 along n fields?	$sfp_n(v_2, v_1) \vee$ $\exists\, u, w \colon \begin{pmatrix} sfp_n(u, v_1) \wedge \\ roc_n(u) \wedge n(u, w) \\ \wedge\, sfp_n(v_2, w) \end{pmatrix}$
$r_{n,x}(v)$	Is v reachable from pointer variable x along n fields?	$\exists\, v_1 \colon x(v_1) \wedge t_n(v_1, v)$
$c_n(v)$	Is v on a directed cycle of n fields?	$\exists\, v_1, v_2 \colon\ roc_n(v_1) \wedge n(v_1, v_2)$ $\wedge\ sfp_n(v, v_2)$
$pr_x(v)$	Does v lie on an sfe_n path from x (does v **pr**ecede x on an n-path to a roc_n node)?	$\exists\, v_1 \colon x(v_1) \wedge sfp_n(v_1, v)$
$pr_{is}(v)$	Does v lie on an sfe_n path from a shared node (does v **pr**ecede a shared node on an n-path to a roc_n node)?	$\exists\, v_1 \colon is_n(v_1) \wedge sfp_n(v_1, v)$

Table 3. Defining formulas of instrumentation relations. The sharing relation is_n is defined as in Tab. 2. Relations $r_{n,x}$ and c_n are redefined via first-order-logic formulas in terms of other relations.

Defining Directed Spanning Forests Instrumentation relation sfe_n—*sfe* stands for **s**panning-**f**orest **e**dge—is used to maintain the set of edges that form a spanning forest of list nodes. In Hesse's work, the spanning-forest edges retain the direction of the n edges. As a result, he maintains spanning forests, in which the edges lead to the roots of the spanning forest, which are designated as roc_n nodes in our abstraction. For clarity of presentation, we define sfe_n to be the reverse of n edges (all but the edges leaving roc_n nodes). The graph defined by the sfe_n relation then defines a *directed* spanning forest with roc_n nodes as spanning-forest roots and with the usual orientation of spanning-forest edges.

Instrumentation relation sfp_n—*sfp* stands for **s**panning-**f**orest **p**ath—is used to maintain the set of paths in the spanning forest of list nodes. Binary reachability in the actual lists (see relation t_n in Tab. 3) can be defined in terms of n, roc_n, and sfp_n using a first-order-logic formula: v_2 is reachable from v_1 if there is a spanning-forest path from v_2 to v_1 or there is a pair of spanning-forest paths, one from the source of a cut edge (a roc_n node) to v_1 and the other from v_2 to the target of the cut edge (the n-successor of the same roc_n node).

Unary reachability relations $r_{n,x}$ and the cyclicity relation c_n can be defined via first-order formulas, as well. We defined $r_{n,x}$ in terms of binary reachability relation t_n. While we could define c_n in terms of t_n, as well, we chose another simple definition by observing that a node lies on a cycle if and only if there is a spanning-forest path from it to the target of a cut edge (the n-successor of a roc_n node).

Fig. 5(b) shows 2-valued structure S_5, which represents the store of Fig. 1(a) using the extended set of core and instrumentation relations. The relations pr_x and pr_{is} will be explained shortly.

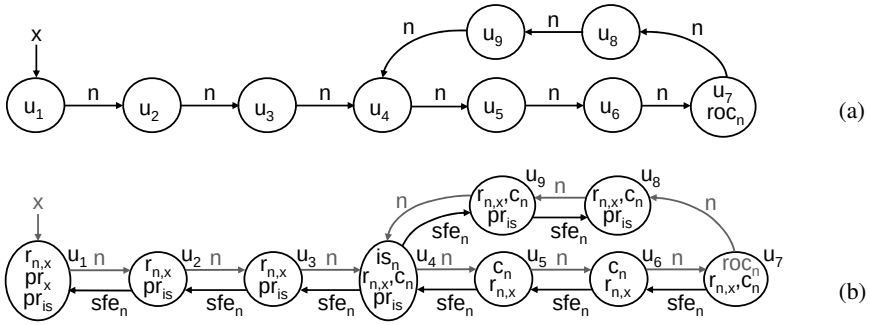

Fig. 5. A logical structure S_5 that represents the store shown in Fig. 1(a) in graphical form: (a) S_5 with the extended set of core relations.(b) S_5 with the extended set of core and instrumentation relations (core relations appear in grey). Transitive-closure relations sfp_n and t_n have been omitted to reduce clutter. The values of the transitive-closure relations can be readily seen from the graphical representation of relations sfe_n and n. For instance, node u_5 is related via the sfp_n relation to itself and all nodes appearing to the left or above it in the pictorial representation.

Preserving Node Ordering on a Cycle in the Presence of Abstraction The fact that our techniques need to be applicable in the presence of abstraction introduces a complication that is not present in the setting studied by Hesse. His concern was with the expressibility of certain properties within the confines of a logic with certain syntactic restrictions. Our concern is with the ability to maintain precision in the framework of canonical abstraction.

Unary reachability relations $r_{n,x}$ (one for every program variable x) play a crucial role in the analysis of programs that manipulate acyclic linked lists. In addition to keeping disjoint lists summarized separately, they keep list nodes that have been visited during a traversal summarized separately from nodes that have not been visited: if x is the pointer used to traverse the list, then the nodes that have been visited will have value 0 for relation $r_{n,x}$, while the nodes that have not been visited will have value 1. If a list contains a cycle, then all nodes on the cycle

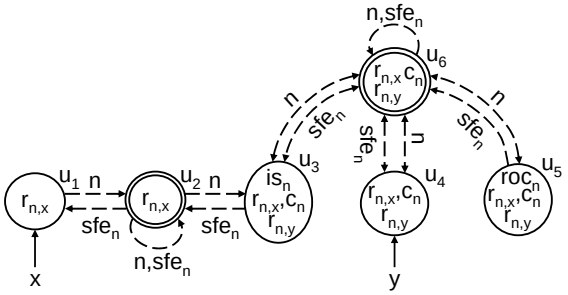

Fig. 6. A 3-valued structure S_6 that is the canonical abstraction of structure S_4 if relations pr_x and pr_{is} are not added to $\mathcal{A}$ and node u_7 is the roc_n node.

are reachable from the same set of variables, namely, all variables that point to any node in that list. As a result, the instrumentation relations discussed thus far cannot prevent nodes u_4, u_6, and u_8 of S_4 shown in Fig. 4 from being summarized together. Thus, assuming that u_7 is the roc_n node, the canonical abstraction of S_4 is the 3-valued structure S_6 shown in Fig. 6. The nodes represented by u_4, u_6, and u_8 of S_4 are represented by the single summary individual u_6 in S_6. The symmetry hides all information about the order of traversal via pointer variable y. Moreover, the values of the sfp_n relation (not shown in Fig. 6) lose precision because ancestors of the shared node in the spanning tree are summarized together with its descendants in the spanning tree.

We break the symmetry of the nodes on a cycle using a general mechanism via unary properties akin to unary reachability relations $r_{n,x}$. In the definitions of relations pr_x of Tab. 3, full reachability (relation t_n) has been replaced with spanning-forest reachability (relation sfp_n). The relations pr_x distinguish nodes according to whether or not they are reachable from program variable x along spanning-forest edges. The relation pr_{is} is defined similarly but using instrumentation relation is_n; pr_{is} partitions the nodes of a panhandle list into ancestors and descendants of the shared node in the spanning tree. Fig. 7 shows structure S_7 that is the canonical abstraction of S_4 of Fig. 4, assuming that u_7 is the roc_n node. In S_7, each of the nodes u_4, u_6, and u_8 has a distinct vector of values for the relations pr_y and pr_{is}, thus breaking the symmetry.

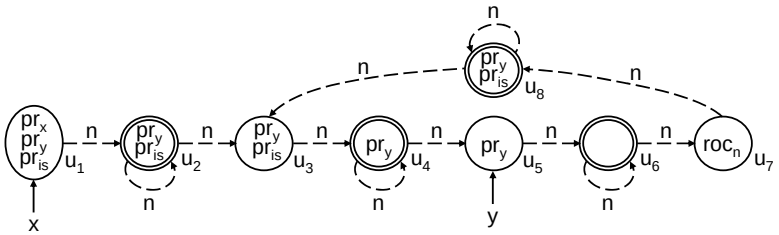

Fig. 7. A 3-valued structure S_7 that is the canonical abstraction of structure S_4 if node u_7 is the roc_n node. S_7 represents panhandle lists of type XY, such as the store of Fig. 1(b). The only instrumentation relations shown in the figure are pr_x, pr_y, and pr_{is}. As in structure S_4 shown in Fig. 4, $r_{n,x}$ holds for all nodes, $r_{n,y}$ and c_n hold for all nodes on the cycle, and is_n holds for u_3.

Automatic Generation of Maintenance Formulas for Instrumentation Relations In his thesis, Hesse gives hand-specified update formulas for a collection of relations that are used for maintaining a spanning-forest representation of possibly-cyclic linked lists. Instead of specifying them by hand, we rely on finite differencing to generate relation-maintenance formulas for all instrumentation relations. Finite-differencing-generated maintenance formulas have been effective in maintaining all relations defined via first-order-logic formulas, i.e., all relations of Tab. 3 except sfp_n. Additionally, under certain conditions, finite-differencing-generated maintenance formulas have been effective in maintaining relations defined via the reflexive transitive closure of binary relations. The necessary conditions for this technique to be applicable for the maintenance of relation sfp_n are:

Acyclicity condition: the graph defined by sfe_n needs to be acyclic;
Unit-size-change condition: the change to the graph effected by any program statement needs to be a single-edge addition or deletion (but not both).

The acyclicity condition applies in our setting because the graph defined by sfe_n defines a spanning forest. The unit-size-change condition requires some discussion.

The relation sfe_n is defined in terms of n and roc_n. While we have not yet discussed the relation-update formulas for core relation roc_n, it should be clear that the value of the relation roc_n should only change in response to a change in the value of a node's `n` field. There are two types of statements that change the value of the `n` field and thus may have an effect that should be reflected in the value of the sfe_n relation, namely, statements of the forms `x->n = NULL` and `x->n = y`. The former destroys the n edge leaving the node pointed to by `x`, and the latter creates a new n-connection from the node pointed to by `x` to the node pointed to by `y`. While both of these statements add or remove a single edge of the n relation, it is not necessarily the case that they add or remove a single edge of the sfe_n relation. When interpreted on logical structure S_7 of Fig. 7, statement `y->n = NULL` has the effect of deleting the n edge leaving u_5, an action that should result in the deletion of the sfe_n edge entering u_5 (not shown in the figure). However, to preserve the spanning-forest representation, we need to ensure that roc_n holds only for nodes that lie on a cycle and that sfe_n represents spanning-forest edges. This requires setting the value of roc_n for u_7 to 0 and adding an sfe_n edge from u_8 to u_7. Because, as this example illustrates, a language statement may result in the deletion of one sfe_n edge and the addition of another, our technique for maintaining instrumentation relations defined via the transitive-closure operator does not apply.

To work around this problem, we apply each transformer associated with statements `x->n = NULL` and `x->n = y` in two phases. In one phase, we apply the part of the transformer that corresponds to the relation n and reflect it in the values of all instrumentation relations. In the other phase, we apply the part of the transformer that corresponds to the relation roc_n and reflect it in the values of all instrumentation relations. As we explain below, each phase of the two transformers satisfies the requirement that the change adds a single edge or removes a single edge of the sfe_n relation.[6] Additionally, by paying attention to the order of phases, we ensure that the graph defined by the relation sfe_n remains acyclic throughout the application of the transformers.

To preserve the acyclicity condition in the case of statement `x->n = NULL`, we apply the part of the transformer that corresponds to the relation n first:

$$n'(v_1, v_2) = n(v_1, v_2) \wedge \neg x(v_1). \tag{2}$$

Unless `x` points to a roc_n node (or `x->n` is `NULL`), this phase results in the deletion of the sfe_n edge that enters the node pointed to by `x`. In the second phase, we apply the part of the transformer that corresponds to the relation roc_n:

$$roc'_n(v) = roc_n(v) \wedge \exists\, v_1 \colon n(v, v_1) \wedge sfp_n(v, v_1) \tag{3}$$

[6] The third problem with defining roc_n as an instrumentation relation (alluded to earlier in the section) is that we would lose the ability to apply the two parts of a transformer separately: the change in the values of n would immediately trigger a change in the values of roc_n. The resulting transformer would not be able to satisfy the unit-size-change condition.

This phase sets the roc_n property of the source n_s of a cut edge to 0, if there is no longer a spanning-forest path from n_s to the target n_t of the same cut edge. When this happens and `x` does not point to n_s, i.e., the cut edge is not being deleted, this phase results in the addition of an sfe_n edge from n_t to n_s.

To preserve the acyclicity condition in the case of statement `x->n = y`, we apply the part of the transformer that corresponds to the relation roc_n first:

$$roc'_n(v) = roc_n(v) \vee (x(v) \wedge \exists\, v_1 \colon y(v_1) \wedge sfp_n(v, v_1)) \tag{4}$$

If there is a spanning-forest path from node n_x, pointed to by `x`, to node n_y, pointed to by `y`, the statement creates a new cycle in the data structure. The update of Formula (4) sets the roc_n property of n_x to 1, thus making n_x the source of a new cut edge and n_y the target of the cut edge. Because there was no n edge from n_x to n_y prior to the execution of this statement,[7] this phase results in no change to the sfe_n relation. In the second phase, we apply the part of the transformer that corresponds to the relation n:

$$n'(v_1, v_2) = n(v_1, v_2) \vee (x(v_1) \wedge y(v_2)) \tag{5}$$

Unless the node pointed to by `x` became a roc_n node in the first phase, this phase results in the addition of an sfe_n edge from n_y to n_x.

The break-up of the transformers corresponding to statements `x->n = NULL` and `x->n = y` into two phases, as described above, ensures that the sfe_n relation remains acyclic throughout the analysis (the acyclicity condition) and that the change to the sfe_n relation effected by each phase is a unit-size change (the unit-size-change condition).[8] Thus, it is sound to maintain sfp_n ($= sfe_n^*$) via the techniques described in [14]. Additionally, it is also sound to maintain the remaining instrumentation relations via those techniques because the remaining relations are defined by first-order-logic formulas. Soundness guarantees that the stored values of instrumentation relations agree with the relations' defining formulas throughout the analysis. However, the stored values may not agree with the relations' *intended meanings*. For instance, if the `n`-transfer phase of the transformer for statement `x->n = NULL` removes a non-cut `n` edge on a cycle, the sfe_n relation will temporarily not span the entire list. However, as long as we do not query the results of abstract interpretation between the phases of a two-phase transformer, the stored values of instrumentation relations agree with the relations' intended meanings, as well as their defining formulas.

Optimized Maintenance of Relation sfp_n By demonstrating that the acyclicity and unit-size-change conditions hold for relation sfe_n, we were able to rely on the techniques of [14] to maintain the relation sfp_n. Note, however, that the definition of sfe_n

[7] By normalizing procedures to include a statement of the form `x->n = NULL` prior to a statement of the form `x->n = y`, we ensure that `x->n` is always `NULL` prior to the latter assignment.

[8] Ensuring the unit-size-change condition requires answering a question that is in general undecidable. However, we found that a conservative approximation based on a syntactic analysis of logical formulas suffices for the types of analyses we have performed so far [14].

ensures that the graph defined by sfe_n is not only acyclic but is tree-shaped. This knowledge has no bearing on the maintenance formulas that reflect a positive unit-size change $\Delta^+[sfe_n]$ to the sfe_n relation in the values of the sfp_n relation (see [14, Formula 8]). However, it allows a negative unit-size change $\Delta^-[sfe_n]$ to the sfe_n relation to be reflected in the values of the sfp_n relation in a more efficient manner. In a tree-shaped graph, there exists at most one path between a pair of nodes; if that path goes through the sfe_n edge to be deleted, it should be removed (cf. [14, Formula 10]):

$$\begin{aligned} sfp'_n(v_1, v_2) = \; & sfp_n(v_1, v_2) \\ & \wedge \neg(\exists\, v'_1, v'_2 \colon sfp_n(v_1, v'_1) \wedge \Delta^-[sfe_n](v'_1, v'_2) \wedge sfp_n(v'_2, v_2)). \end{aligned} \tag{6}$$

We extended our finite-differencing technique with the optimized schema for maintaining the transitive closure of a tree-shaped binary relation in response to a negative unit-size change in the relation. We will refer to the method of [14] as *acyclic-sfe_n maintenance* and the optimized method as *tree-shaped-sfe_n maintenance*.

4 Expressing Properties of Transformations

When discussing properties of `Reverse`, we are interested in making assertions that compare the state of a store at the end of the procedure with its state at the start. For instance, we may be interested in checking that all tree nodes reachable from variable `x` at the start of the procedure are guaranteed to be reachable from `x` at the end. To allow the user to make such assertions, we double the vocabulary: for each relation p, we extend the program-analysis specification with a *history* relation, p^0, which serves as an indelible record of the state of the store at the entry point. We will use the term history relations to refer to the latter kind of relations, and the term *active* relations to refer to the relations from the original vocabulary. We can now express the property mentioned above:

$$\forall\, v \colon r_{n,x}(v) \Leftrightarrow r^0_{n,x}(v). \tag{7}$$

If Formula (7) evaluates to 1, then the elements reachable from `x` after the procedure executes are exactly the same as those reachable at the beginning of the procedure.

In addition to history relations, we introduce a collection of nullary instrumentation relations that track whether active relations have changed from their initial values. For each active relation $p(v_1, \ldots, v_k)$, the relation $same_p()$ is defined by formula $\forall\, v_1, \ldots, v_k \colon p(v_1, \ldots, v_k) \Leftrightarrow p^0(v_1, \ldots, v_k)$. We can now use $same_{r_{n,x}}()$ in place of Formula (7). Additionally, we introduce a unary relation ch_n which tracks the changes to the sole binary core relation, n. The relation ch_n is defined by the formula $ch_n(v) = \neg\forall\, v_1 \colon n(v, v_1) \Leftrightarrow n^0(v, v_1)$; it is *not* part of the set of abstraction relations, $\mathcal{A}$.

5 In-Situ List-Reversal Algorithm

Fig. 8 shows the list-reversal algorithm that we analyze. The algorithm performs the reversal in place using three pointer variables, `x`, `y`, and `t`. The `n` field of list nodes is reversed on lines [7] and [8]. During the execution of the statements on those lines, `x` points to the next node to be processed, `y` points to the node whose `n` field is reversed, and `t` points to the predecessor of that node.

```
[1]  void reverse(List *x)
[2]  { List *y = NULL;
[3]    while (x != NULL) {
[4]      t = y;
[5]      y = x;
[6]      x = x->n;
[7]      y->n = NULL;
[8]      y->n = t;
[9]    }
[10]   x = y;
[11] }
```

Fig. 8. In-situ list reversal algorithm

First, let us consider how `Reverse` processes an acyclic list L_a with head u_1, pointed to by `x`. Fig. 9 shows a logical structure S_9 that represents a store that arises before line [7] during the application of `Reverse` to L_a. At this point the `n` edges of nodes $u_1, \ldots, u_3$ have been reversed, while the remaining edges retain their original orientation. The statements on lines [7] and [8] replace the `n` edge from u_4 to u_5 with an `n` edge from u_4 to u_3. The traversal continues until, on the last loop iteration, `t` is set to point to u_7's predecessor in the input list, `y` is set to point to u_7, and `x` is set to `NULL`. The subsequent execution of lines [7] and [8] reverses the remaining `n` edge. The head of the reversed list is u_7, pointed to by `y`. As in the input list, no node lies on a cycle. The last statement of the procedure (the assignment on line [10]) restores `x` as the head pointer. The transformation described above can be stated formally using history relations as follows:

$$same_{r_{n,x}}() \wedge same_{c_n}() \wedge \forall\, v_1, v_2 \colon n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1). \tag{8}$$

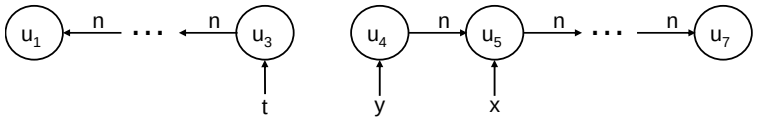

Fig. 9. Logical structure S_9 that represents a store that arises prior to line [7] of `Reverse` when the algorithm is applied to an acyclic list.

Let us consider how `Reverse` processes a list L_c that consists of a single cycle without a panhandle, such as the acyclic list L_a discussed above, but with an additional `n` edge from u_7 to u_1. The behavior of `Reverse` on list L_c is nearly identical to its behavior on list L_a. The outgoing `n` edges are reversed one at a time until, on the last iteration, `t` is set to point to u_7, `y` is set to point to u_1, and `x` is set to `NULL`. The subsequent execution of lines [7] and [8] reverses the remaining `n` edge from u_7 to u_1. The head of the reversed list remains u_1, pointed to by `y`. Every list node still lies on a cycle. The last statement of the procedure (the assignment on line [10]) restores `x` as the head pointer. The transformation of lists such as L_c also obeys the property specified in Formula (8).

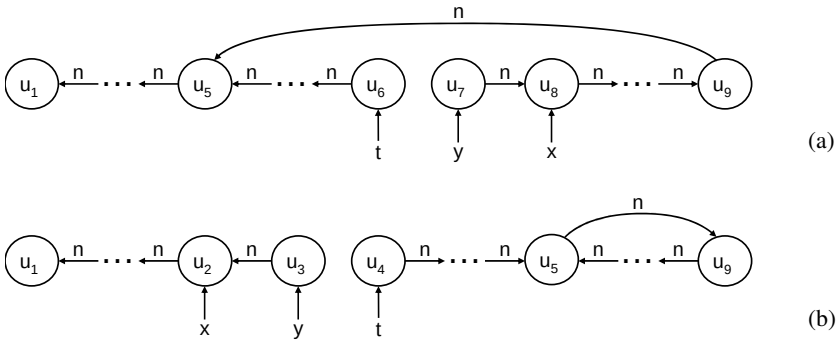

Fig. 10. Logical structures that represent stores that arise prior to line [7] of `Reverse` when the algorithm is applied to a panhandle list. (a) Logical structure that represents a store that arises while `Reverse` processes nodes that lie on the cycle, i.e., after processing nodes that lie in the panhandle once. (b) Logical structure that represents a store that arises while `Reverse` processes nodes that lie on the panhandle for the second time, i.e., after processing nodes that lie on the cycle.

Now, we discuss how `Reverse` processes a panhandle list L_p. Initially, the procedure advances the three pointer variables, `x`, `y`, and `t`, down the panhandle, reversing the `n` edges out of `y`. After the panhandle is processed, the algorithm proceeds with the processing of the cycle. Fig. 10(a) shows a logical structure that represents a store that arises prior to line [7] while `Reverse` processes nodes that lie on the cycle. Until `Reverse` completes the processing of the cycle, the steps are identical to the steps taken during the processing of lists L_a and L_c. Note that the orientation of the `n` edges in the panhandle is reversed when the loop body is executed with `x` pointing to u_5 (while reversing the backedge at the end of processing the cycle). As a result, the algorithm proceeds along the reversed `n` edges down the panhandle, reestablishing the original orientation of those edges. Fig. 10(b) shows a logical structure that represents a store that arises prior to line [7] while `Reverse` processes panhandle nodes for the second time. Instead of reversing every `n` edge in the list, as it does for lists L_a and L_c,[9] the algorithm reverses the direction of every `n` edge on the cycle but reestablishes the original direction of the `n` edges in the panhandle. The cyclicity property of all nodes remains as it was on input. The head of the output list remains u_1, pointed to by `y`. The last statement of the procedure (the assignment on line [10]) restores `x` as the head pointer. The transformation described above can be stated formally using history relations as follows:

$$\begin{aligned} & & same_{r_{n,x}}() \wedge same_{c_n}() \wedge \\ \forall\, v_1, v_2 \colon & \quad (c_n^0(v_1) \wedge c_n^0(v_2)) \wedge (n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)) \\ & \vee \neg(c_n^0(v_1) \wedge c_n^0(v_2)) \wedge (n(v_1, v_2) \Leftrightarrow n^0(v_1, v_2)). \end{aligned} \tag{9}$$

Note that while the behavior of `Reverse` on lists consisting of a cycle without a panhandle can be described by Formula (8), as we mentioned above, it can also be described by Formula (9). (The case described by formula $\neg(c_n^0(v_1) \wedge c_n^0(v_2))$ never arises.)

[9] Reversing every `n` edge of a panhandle list is not possible because it requires the shared node (u_5 in Fig. 10) to have two outgoing `n` edges.

6 Establishing Properties of `Reverse`

In this section we describe how the abstraction-refinement mechanism presented in [10] can be used to verify automatically that `Reverse` obeys the properties described in the previous section.

Constructing All Valid Inputs for `Reverse` To verify that `Reverse` satisfies the properties discussed in the previous section, we need a collection of 3-valued abstract input structures that represent all valid inputs to the procedure. Our methodology for obtaining values for abstract input structures is to perform an abstract interpretation on a loop that nondeterministically constructs the family of all valid inputs to the program (we call such a loop a **D**ata-**S**tructure **C**onstructor, or DSC). This allows the values of instrumentation relations to be maintained (as input structures are manufactured from the empty store) rather than computed; in general, this results in more precise values for the instrumentation relations without requiring the user to specify input 3-valued logical structures.

```
[1]  List *x = NULL;
[3]  int sz =
[4]    sizeof(List);
[5]  while (?) {
[6]    List *t = malloc(sz);
[11]   t->n = x;
[12]   x = t;
[13] }
```

(a)

```
List *x, *y, *h;                [1]
x = y = h = NULL;               [2]
int sz =                        [3]
  sizeof(List);                 [4]
while (?) {                     [5]
  List *t = malloc(sz);         [6]
  // save the last node         [7]
  if (y == NULL) y = t;         [8]
  // save a node (or NULL)      [9]
  if (?) h = t;                [10]
  t->n = x;                    [11]
  x = t;                       [12]
}                              [13]
// if y and h are non-NULL,    [14]
// this will create a cycle    [15]
if (y != NULL) y->n = h;       [16]
```

(b)

Fig. 11. (a) The Data-Structure Constructor for acyclic linked lists. (b) The Data-Structure Constructor for possibly-cyclic linked lists (including acyclic and panhandle lists). The differences between the two versions appear in bold.

Two examples of our methodology are depicted in Fig. 11. The loop on the left nondeterministically constructs an acyclic linked list pointed to by `x`: a list is constructed from tail to head (i.e., most deeply nested node first); the loop exits after some number of nodes have been added at the front of the list. The slight modification shown on the right nondeterministically constructs a (cyclic or acyclic) linked list pointed to by `x`. This is achieved by setting `y` to point to the last list node on line [8], nondeterministically setting `h` to point to some list node (or `NULL`) on line [10], and setting `y->n` to point to `h` on line [16] if `y` is non-`NULL` (possibly completing a cycle). If `h` is `NULL`, the

DSC constructs an acyclic list. If h points to the head of the list, the DSC constructs a list consisting of a cycle with no panhandle. If h is neither NULL nor points to the head of the list, the DSC constructs a panhandle list.

Abstract interpretation of the DSC of Fig. 11(b) constructs an abstract representation of all linked lists pointed to by x. When testing the application of a procedure to acyclic lists, we select only those structures collected at the exit of the DSC that satisfy the following formula:

$$(\exists\, v\colon\, r_{n,x}(v)) \wedge (\forall v\colon\, r_{n,x}(v) \Rightarrow \neg c_n(v)) \tag{10}$$

We will refer to input abstractions satisfying Formula (10) as *type Acyclic*. When testing the application of a procedure to cyclic lists without a panhandle, we select only those structures collected at the exit of the DSC that satisfy the following formula:

$$(\exists\, v\colon\, r_{n,x}(v)) \wedge (\forall v\colon\, r_{n,x}(v) \Rightarrow c_n(v)) \tag{11}$$

We will refer to input abstractions satisfying Formula (11) as *type Cyclic*. When testing the application of a procedure to panhandle lists, we select only those structures collected at the exit of the DSC that satisfy the following formula:

$$(\exists\, v_1\colon\, r_{n,x}(v_1) \wedge \neg c_n(v_1)) \wedge (\exists\, v_2\colon\, r_{n,x}(v_2) \wedge c_n(v_2)) \tag{12}$$

We will refer to input abstractions satisfying Formula (12) as *type Panhandle*. Note that Formulas (10)–(12) ensure that each of the input types admits only non-empty lists. Note also that the three types represent disjoint collections of data structures. Additionally, the cross product of the set of lists represented by type *Acyclic* and the set of lists represented by type *Cyclic* is in a one-to-one correspondence with the set of lists represented by type *Panhandle:* the acyclic-list component corresponds to the panhandle of a panhandle list and the cyclic-list component corresponds to its cycle. We will make use of these facts in §7.

In addition to constructing the valid inputs prior to the first analysis of Reverse, the DSC is used for constructing *refined* inputs on every iteration of abstraction refinement: after abstraction refinement introduces additional instrumentation relations, the abstract interpretation of the DSC is performed using an extended vocabulary that contains the new relation symbols; the 3-valued structures collected at the exit node of the DSC become the abstract input to the original procedure for the subsequent abstract interpretation of the procedure.

Note that history relations (such as $r^0_{n,x}(v)$ from §4) are intended to record the state of the store at the entry point to the procedure or, equivalently, at the exit from the DSC. To make sure that these relations have appropriate values, they are maintained in tandem with their active counterparts during abstract interpretation of the DSC. When abstract input refinement is completed, values of history relations are frozen in preparation for the abstract interpretation that is about to be performed on the procedure proper.

Abstraction-Refinement Steps After an abstraction of the appropriate valid input is constructed by analyzing the DSC, the abstract interpretation collects all structures that arise at all program points of Reverse. To check if Reverse satisfies the expected

properties, we check if all structures collected at the exit of `Reverse` satisfy the appropriate query (Formula (8) when testing the application of the procedure to lists represented by type *Acyclic* and Formula (9) when testing the application of the procedure to lists represented by type *Panhandle*; we can check either query when testing the application of the procedure to lists represented by type *Cyclic*).

Both queries (Formulas (8) and (9)) contain formula $n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$ as a subformula. Because this formula evaluates to $1/2$ under any assignment that maps v_1 and v_2 to the same summary individual with a $1/2$-valued self-loop for the relation n, it should come as no surprise that the first run of abstract interpretation returns an indefinite answer, whether we are checking Formula (8) or Formula (9).

In [10], we introduced *subformula-based refinement*, which analyzes the sources of imprecision in the evaluation of the query in a structure collected at the exit of a procedure, and chooses how to define new instrumentation relations using subformulas of the query that contribute to the indefinite answer. Tab. 4 shows the instrumentation relations that are introduced by subformula-based refinement after Formula (8) evaluates to $1/2$ on a structure collected at the exit of `Reverse`, given an input abstraction of either type *Acyclic* or *Cyclic*. Column 2 of Tab. 4 shows the imprecise subformulas that are used to define new instrumentation relations. To gain precision improvements from storing and maintaining the new instrumentation relations all occurrences of the defining formulas for the new instrumentation relations in the query and in the definitions of other instrumentation relations are replaced with the use of the corresponding new instrumentation-relation symbols. Here, the use of Formula (8) in the query is replaced with the use of the stored value $rev_1()$. Then the definitions of all instrumentation relations are scanned for occurrences of the defining formulas for $rev_1, \ldots, rev_6$. These occurrences are replaced with the names of the six relations. In this case, only the new relations' definitions are changed, yielding the definitions given in Column 3 of Tab. 4.

Relation	Imprecise Subformula	Defining formula
$rev_1()$	$same_{r_{n,x}}() \wedge same_{c_n}() \wedge$ $\forall v_1, v_2 \colon n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$	$same_{r_{n,x}}() \wedge same_{c_n}() \wedge rev_2()$
$rev_2()$	$\forall v_1, v_2 \colon n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$	$\forall v_1 : rev_3(v_1)$
$rev_3(v_1)$	$\forall v_2 \colon n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$	$\forall v_2 : rev_4(v_1, v_2)$
$rev_4(v_1, v_2)$	$n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$	$rev_5(v_1, v_2) \wedge rev_6(v_2, v_1)$
$rev_5(v_1, v_2)$	$n(v_1, v_2) \Rightarrow n^0(v_2, v_1)$	$n(v_1, v_2) \Rightarrow n^0(v_2, v_1)$
$rev_6(v_2, v_1)$	$n^0(v_2, v_1) \Rightarrow n(v_1, v_2)$	$n^0(v_2, v_1) \Rightarrow n(v_1, v_2)$

Table 4. Instrumentation relations created by subformula-based refinement when the application of `Reverse` is checked against the query expressed in Formula (8) on an input abstraction of either type *Acyclic* or *Cyclic*.

Tab. 5 shows the instrumentation relations that are introduced by subformula-based refinement after Formula (9) evaluates to $1/2$ on a structure collected at the exit of `Reverse`, given an input abstraction of either type *Panhandle* or *Cyclic*. Column 2 of Tab. 5 shows the imprecise subformulas that are used to de-

fine new instrumentation relations. Note that subformulas of $acycSame(v_1, v_2)$, i.e., $\neg(c_n^0(v_1) \wedge c_n^0(v_2)) \wedge (n(v_1, v_2) \Leftrightarrow n^0(v_1, v_2))$ were not introduced. This is because refinement was triggered by imprecise evaluation on a structure that had a single concrete individual in the panhandle. However, relation rev_3 is capable of maintaining the key property of nodes in the panhandle with enough precision, so that another refinement iteration is not required. Column 3 of Tab. 5 gives the definitions of the new instrumentation relations after all occurrences of the defining formulas of new instrumentation relations in the query and in the definitions of other instrumentation relations have been replaced with the use of the corresponding new instrumentation-relation symbols. Again, only the query and new relations' definitions are changed.

Relation	Imprecise Subformula	Defining formula
$rev_1()$	$same_{r_{n,x}}() \wedge same_{c_n}() \wedge$ $\forall v_1, v_2 :\ cycRev(v_1, v_2) \vee acycSame(v_1, v_2)$	$same_{r_{n,x}}() \wedge same_{c_n}() \wedge rev_2()$
$rev_2()$	$\forall v_1, v_2 :\ cycRev(v_1, v_2) \vee acycSame(v_1, v_2)$	$\forall v_1 : rev_3(v_1)$
$rev_3(v_1)$	$\forall v_2 :\ cycRev(v_1, v_2) \vee acycSame(v_1, v_2)$	$\forall v_2 : rev_4(v_1, v_2)$
$rev_4(v_1, v_2)$	$cycRev(v_1, v_2) \vee acycSame(v_1, v_2)$	$rev_5(v_1, v_2) \vee acycSame(v_1, v_2)$
$rev_5(v_1, v_2)$	$cycRev(v_1, v_2)$	$(c_n^0(v_1) \wedge c_n^0(v_2)) \wedge rev_6(v_2, v_1)$
$rev_6(v_1, v_2)$	$n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$	$rev_7(v_1, v_2) \wedge rev_8(v_2, v_1)$
$rev_7(v_1, v_2)$	$n(v_1, v_2) \Rightarrow n^0(v_2, v_1)$	$n(v_1, v_2) \Rightarrow n^0(v_2, v_1)$
$rev_8(v_2, v_1)$	$n^0(v_2, v_1) \Rightarrow n(v_1, v_2)$	$n^0(v_2, v_1) \Rightarrow n(v_1, v_2)$

Table 5. Instrumentation relations created by subformula-based refinement when the application of `Reverse` is checked against the query expressed in Formula (9) on an input abstraction of either type *Panhandle* or *Cyclic*. For compactness, we refer to formula $(c_n^0(v_1) \wedge c_n^0(v_2)) \wedge (n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1))$ as $cycRev(v_1, v_2)$ and to formula $\neg(c_n^0(v_1) \wedge c_n^0(v_2)) \wedge (n(v_1, v_2) \Leftrightarrow n^0(v_1, v_2))$ as $acycSame(v_1, v_2)$.

After the introduction of the new instrumentation relations (Tab. 4 or 5, depending on the query being verified), the abstract interpretation of the DSC is performed using an extended vocabulary that contains the new instrumentation-relation symbols. The subsequent abstract interpretation of `Reverse` succeeds: in all of the structures collected at the exit, $rev_1() = 1$.

Establishing that `Reverse` Terminates We can establish that `Reverse` terminates using a few unary core relations and a simple progress monitor. We introduce a collection of unary core *state relations*, $state_0(v)$, $state_1(v)$, and $state_2(v)$.[10] Every time the reversal of the `n` pointer of the list node pointed to by `y` is completed (after line [8] of Fig. 8), the node's state is changed to the next state. (The state relations carry no semantics with respect to the pointer values of nodes; they simply record the "visit counts" for each node.) For each state relation s, we create a copy of s, which is used to save the values of relation s at the start of the currently-processed loop iteration (after line [3] of Fig. 8). We give the new relations the superscript lh to indicate that they hold the

[10] The state relations are *not* added to the set of abstraction relations, $\mathcal{A}$.

loop-head values. The first abstract operation of each iteration of the loop takes a snapshot of the current states of nodes: $state_i^{lh}(v) \leftarrow state_i(v)$, for each $i \in [0..2]$ and each assignment of v to an individual in the abstract structure being processed. Additionally, it asserts that `x` does not point to a list node in state 2 at the head of the loop (at that point, `x` points to the node whose `n` edge is about to be reversed). The last operation of every loop iteration performs a progress test by asserting the following formula:

$$\begin{aligned}\exists\, v\colon\ & \big(state_0^{lh}(v) \wedge state_1(v) \vee state_1^{lh}(v) \wedge state_2(v)\big) \\ \bigwedge\ \forall\, v_1 \neq v\colon\ & \textstyle\bigwedge_{i\in[0..2]}\ \big(state_i^{lh}(v_1) \Leftrightarrow state_i(v_1)\big)\end{aligned}$$

The assertion ensures that one node's state makes forward progress (the first line of the assertion) and that no other node changes state (the second line of the assertion). Together with the assertion that `x` does not point to a list node in state 2 at the start of the loop, the above progress monitor establishes that each list node is visited at most twice, thus establishing that the algorithm terminates.

7 Performance

The tables shown in Fig. 12 give execution times that were collected on a 3GHz Linux PC. The rows indicate the type of data structures assumed as input, and the columns indicate the query to be verified. In each case, one round of abstraction refinement was required to obtain the definite answer 1 to the query. In other words, two rounds of analysis were performed for both the DSC and `Reverse`: the first analysis round of the DSC and `Reverse` used the initial abstraction (the core relations of Tab. 1, core relation roc_n, the instrumentation relations of Tab. 3, and the history relations of §4), while the second round used the final abstraction, which additionally included the relations of Tab. 4 or 5, depending on the query. For a given abstraction, the cost of the DSC analysis is nearly identical for all input types because the general DSC of Fig. 11(b) constructs an abstraction of all input types, from which structures that represent the chosen input type are selected at the end using Formula (10), (11), or (12). To gain a better understanding of the cost of verifying `Reverse` proper, the tables also include the execution times for the last analysis round (using the final abstraction) of `Reverse`, excluding the analysis time for the DSC.

The tables of Fig. 12 show that the use of tree-shaped-sfe_n maintenance in place of acyclic-sfe_n maintenance for maintaining the relation sfp_n results in a reduction of the total analysis time by a factor in the range of 2.8-4.8. The highest-cost analyses are those that include type *Panhandle* as input. Using tree-shaped-sfe_n maintenance, the last iteration of the analysis of `Reverse` with the input abstraction of type *Panhandle*/*Cyclic* takes approximately 2.5 minutes (the total execution time is approximately 4.5 minutes). The last iteration of the analysis of `Reverse` when the input abstraction is of any other type takes under 13 seconds. The majority of the total analysis cost in those cases is due to the use of the general DSC, which could be specialized to produce input abstractions of type *Acyclic* or *Cyclic* more efficiently (e.g., using the DSC shown in Fig. 11(a)).

The two tables of Fig. 12 share many qualitative characteristics. Below we draw some conclusions from Fig. 12(b), but all of the conclusions can be drawn from Fig. 12(a)

Input Type	Query	
	acyclic total/last	panhandle total/last
Acyclic	161.2/11.8	
Cyclic	208.1/28.2	232.9/34.6
Acyclic/*Cyclic*	219.7/38.6	
Panhandle		1320.3/782.3
Panhandle/*Cyclic*		1249.3/810.3

(a)

Input Type	Query	
	acyclic total/last	panhandle total/last
Acyclic	57.1/4.6	
Cyclic	65.6/8.4	71.5/9.0
Acyclic/*Cyclic*	69.1/12.5	
Panhandle		277.1/147.8
Panhandle/*Cyclic*		268.1/154.9

(b)

Fig. 12. Execution times in seconds using (a) acyclic-sfe_n maintenance for maintaining the relation sfp_n; (b) tree-shaped-sfe_n maintenance for maintaining the relation sfp_n. In row labels, input types "*Acyclic*/*Cyclic*" and "*Panhandle*/*Cyclic*" denote an abstraction that represents lists of either type. The label of column 2 (query "acyclic") denotes the query of Formula (8). The label of column 3 (query "panhandle") denotes the query of Formula (9). Empty cells indicate inappropriate input/query combinations. The first number in each column represents the total execution time for all iterations of the analysis (on both the DSC and `Reverse`). The second number represents the execution time for only the last iteration of the analysis of `Reverse` (and not the DSC).

equally well. As expected, the cost of the last run of the analysis of `Reverse` when the input abstraction is of type *Acyclic*/*Cyclic* is close to the sum of the cost when the input abstraction is of type *Acyclic* and the cost when the input abstraction is of type *Cyclic*. Similarly, the cost of the last run of the analysis of `Reverse` when the input abstraction is of type *Panhandle*/*Cyclic* is close to the sum of the cost when the input abstraction is of type *Panhandle* and the cost when the input abstraction is of type *Cyclic*. Curiously, the total cost of the analysis when the input abstraction is of type *Panhandle* is slightly higher than the total cost when the input abstraction is of type *Panhandle*/*Cyclic*. The reason is that a structure of type *Cyclic* triggers the refinement process at an earlier point. The resulting shorter execution of the first run of the analysis of `Reverse` explains the counterintuitive relation of total execution times. The cost of the analysis when the input abstraction is of type *Cyclic* (both total cost and the cost of the last iteration of the analysis of `Reverse`) is similar for the two queries. The panhandle query (Formula (9)) results in the introduction of a more complex abstraction (cf. Tabs. 5 and 4), so the costs in column 3 of Fig. 12(b) are slightly higher.

The cost of verifying that `Reverse` terminates is negligible (when compared to the cost of verifying the query) because the progress monitor does not increase the size of the reachable state space.

The three analyses represented by column 3 of Fig. 12(a), i.e., analyses using the panhandle query (Formula (9)) and acyclic-sfe_n maintenance, used a maximum of approximately 170 MB of memory, as reported by the Java Runtime. All other analyses required significantly less memory.

As a sanity check, we studied the number of distinct 3-valued structures collected at all points of `Reverse` during the last run of the analysis. As we expected, that information is identical when the analysis relies on acyclic-sfe_n maintenance and when it relies on tree-shaped-sfe_n maintenance, thus providing a cross-validation of the implementation of the two methods. The structure counts are shown in Tab. 6. The table

shows that when the input abstraction is of type *Cyclic*, the same number of structures is collected with either query. Also, the number of structures collected when the input abstraction is of type *Acyclic*/*Cyclic* is the sum of the number when the input abstraction is of type *Acyclic* and the number when the input abstraction is of type *Cyclic*. Similarly, the number of structures collected when the input abstraction is of type *Panhandle*/*Cyclic* is the sum of the number when the input abstraction is of type *Panhandle* and the number when the input abstraction is of type *Cyclic*.

Additionally, we used the data collected in our experiments to answer an instance of the following general question: "Can we predict how much work needs to be done for analysis X when we know how much work is done for related analyses Y and Z?" Given the correspondence of lists represented by type *Panhandle* with combinations of lists represented by type *Acyclic* and lists represented by type *Cyclic*, we made a prediction about the number of structures collected during the analysis of `Reverse` when the input abstraction is of type *Panhandle* (using the panhandle query) based on the number of structures collected during the analyses of `Reverse` when the input abstraction is of types *Acyclic* and *Cyclic* (using the acyclic query). Let a_n, c_n, and p_n, represent the numbers of structures collected at CFG node n during the analysis of `Reverse` when the input abstraction is of type *Acyclic*, *Cyclic*, and *Panhandle*, respectively. For a CFG node n that lies outside the loop of `Reverse`, we expect that $p_n = a_n * c_n$. For a CFG node n that lies inside the loop, we expect that

Input Type	Query	
	acyclic	panhandle
Acyclic	103	
Cyclic	162	162
Acyclic/*Cyclic*	265	
Panhandle		921
Panhandle/*Cyclic*		1083

Table 6. The number of distinct 3-valued structures collected during the last iteration of the analysis of `Reverse` (and not the DSC). Rows and columns have the same meaning as in Fig. 12.

$$p_n = c_{entry} * a_n + a_{exit} * c_n + c_{exit} * a_n, \tag{13}$$

where c_{entry} is the number of structures at the entry node of `Reverse` when the input abstraction is of type *Cyclic*, a_{exit} is the number of structures collected at the exit of `Reverse` when the input abstraction is of type *Acyclic*, and c_{exit} is the number of structures collected at the exit of `Reverse` when the input abstraction is of type *Cyclic*. The intuition behind the first summand of Formula (13) is that every acyclic structure collected at n (when the input abstraction is of type *Acyclic*) can be extended to c_{entry} panhandle structures at n. These structures represent the states in which the panhandle is being reversed before the cycle is entered. The intuition behind the second summand of Formula (13) is that every cyclic structure collected at n (when the input abstraction is of type *Cyclic*) can be extended to a_{exit} panhandle structures. These structures represent the states in which the cycle is being reversed after the panhandle has been reversed. Finally, the intuition behind the third summand of Formula (13) is that every acyclic structure collected at n can be extended to c_{exit} panhandle structures. These structures represent the states in which the panhandle is being un-reversed after the cycle has been reversed. The summation of predicted values for p_n over the nodes n

of `Reverse` gives 858 structures. This prediction is a little short of the actual number (921). This relatively small discrepancy is probably due to the fact that our prediction for a run using the panhandle query (which leads to the abstraction of Tab. 5) is based on numbers for the right-hand side quantities of Formula (13) gathered from runs that use a slightly different abstraction, namely, Tab. 4. The more complex abstraction introduced when verifying the panhandle query apparently creates a few additional intermediate structures.

Note that the sum of the numbers of structures collected during the analyses when the input abstraction is of types *Acyclic* and *Cyclic* is much lower than the number of structures collected during the analysis when the input abstraction is of type *Panhandle*. The sum of the execution times of the analyses when the input abstraction is of types *Acyclic* and *Cyclic* is also much lower than the execution time of the analysis when the input abstraction is of type *Panhandle*. A possible extension of this work is to infer properties of `Reverse` when applied to input abstraction of type *Panhandle* from properties of `Reverse` when applied to input abstractions of types *Acyclic* and *Cyclic*. To make this possible, we need to find a way to infer properties of heap configurations from properties of components of those configurations. The concept of *local heaps* introduced by Rinetzky et al. is relevant in this line of research [16].

8 Related Work

In §3, we compared our work with that of William Hesse, which is closest in spirit to what is reported here. Prior work that is related to the general concept of finite differencing has been discussed in [14]. Work related to the abstraction-refinement mechanism has been discussed in [10]. In this section, we discuss a few approaches that bear resemblance to ours in that they attempt to translate or simulate a data structure that cannot be handled by some core techniques into one that can.

The idea of using spanning-tree representations for specifying or reasoning about data structures that are "close to trees" is not new. Klarlund and Schwartzbach introduced *graph types*, which can be used to specify some common non-tree-shaped data structures in terms of a spanning-tree *backbone* and regular expressions that specify where non-backbone edges occur within the backbone [7]. Examples of data structures that can be specified by graph types are doubly-linked lists and threaded trees. A panhandle list cannot be specified by a graph type because in a graph type the location of each non-backbone edge has to be defined in terms of the backbone using a regular expression, and a regular expression cannot be used to specify the existence of a backedge to *some* node that occurs earlier in the list. In the PALE project [12], which incorporates work on graph types, automated reasoning about programs that manipulate data structures specified as graph types can be carried out using a decision procedure for monadic second-order logic. Unfortunately, the decision procedure has non-elementary complexity. Additionally, the decision procedure cannot handle 2-vocabulary structures, which the present paper uses to express data-structure transformations (the second vocabulary consists of history relations p^0). An advantage of our approach over that of PALE is that we do not rely on the use of a decision procedure.

Immerman et al. presented *structure simulation*, a technique that broadens the applicability of decision procedures to a larger class of data structures [6]. Under certain conditions, it allows data structures that cannot be reasoned about using decidable logics to be translated into data structures that can, with the translation expressed as a first-order-logic formula. Unlike graph types, structure simulation is capable of specifying panhandle lists. However, this technique shares a limitation of graph types because it relies on decision procedures for automated reasoning about programs.

Manevich et al. specified abstractions (in canonical-abstraction and predicate-abstraction forms) for showing safety properties of programs that manipulate possibly-cyclic linked lists [11]. By maintaining reachability within list segments that are not *interrupted* by nodes that are shared or pointed to by a variable, they are able to break the symmetry of a cycle. The definition of several key instrumentation relations in that work makes use of transitive-closure formulas that cannot be handled precisely by finite differencing. As a result, a drawback of that work is the need to define some relation-maintenance formulas by hand. Another drawback is the difficulty of reasoning about reachability (in a list) from a program variable (see reachability relations $r_{n,x}$ of Tab. 3). Because in [11] reachability in a list has to be expressed in terms of reachability over a sequence of uninterrupted segments, a formula that expresses the reachability of node v from program variable `x` in a list has to enumerate all permutations of other program variables that may act as interruptions on a path from `x` to v in the list.

A number of past approaches to the analysis of programs that manipulate linked lists relied on first-order axiomatizations of reachability information. All of these approaches involved the use of first-order-logic decision procedures. While our approach does not have this limitation, it is instructive to compare our work with those approaches that included mechanisms for breaking the symmetry on a cycle. Nelson defined a set of first-order axioms that describe the ternary reachability relation $r_n(u, v, w)$, which has the meaning: w is reachable from u along n edges without encountering v [13]. The use of this relation alone is not sufficient in our setting because in the presence of abstraction we require unary distinctions (such as the relations pr_x and pr_{is} of §3) to break the symmetry. Additionally, the maintenance of ternary relations is more expensive than the maintenance of binary relations. Lahiri and Qadeer specify a collection of first-order axioms that are sufficient to verify properties of procedures that perform a single change to a cyclic list, e.g., the removal of an element [8]. They also verify properties of in-situ list reversal, albeit under the assumption that the input list is acyclic. (We verify properties of `Reverse` when applied to any linked list, including cyclic and panhandle lists.) They break the symmetry of cycles in a similar fashion to how it is done in [11]: the *blocking cells* of [8] are a subset of the interruptions of [11]. The blocking cells include only the set of *head variables*—program variables that act as heads of lists used in the program. This set has to be maintained carefully by the user to (i) satisfy the system's definition of acceptable (*well-founded*) lists, (ii) allow the system to verify useful postconditions, and (iii) avoid falling prey to the difficulty—that arises in [11]—of expressing reachability in the list. The current mechanism of [8] is insufficient for reasoning about panhandle lists because the set of blocking cells does not include shared nodes. This limitation can be partially addressed by generalizing the set of blocking cells to mimic interruptions of [11] more faithfully. However, this

may make it more difficult to satisfy points (ii) and (iii) stated above. As in our work, Lahiri and Qadeer rely on the insight that reachability information can be maintained in first-order logic. They use a collection of manually-specified update formulas that define how their relations are affected by the statements of the language and the (user-inserted) statements that manage the set of head variables.

Gotsman et al. present an interprocedural shape-analysis algorithm that is capable of checking some properties of programs that manipulate possibly-cyclic linked lists [3]. This algorithm is based on a novel abstract domain that consists of formulas in a decidable fragment of Separation Logic [1, 15]. The analysis relies on carefully hand-crafted inductive predicates and axioms to evaluate some properties of possibly-cyclic linked lists precisely and efficiently. However, because the formulas allowed in the analysis need to come from a decidable fragment of Separation Logic and, furthermore, need to make up a finite abstract domain, the set of properties that can be tested is limited. For instance, their analysis of `Reverse` when applied to a panhandle list verifies memory safety and the absence of memory leaks, but shows neither the partial correctness (the property described by Formula (9)), nor the termination of the algorithm. Although our analysis is also based on logic, our abstract domain consists of sets of (abstracted) logical structures. An advantage of our approach is the ability to test more general properties. Additionally, our approach does not rely on decision procedures.[11]

Lee et al. defined a shape-analysis algorithm that extends the shape graphs of Sagiv et al. [17] with grammars [9]. Grammars are employed in place of instrumentation relations for expressing and maintaining derived properties of data structures. The meaning of a grammar is given by an inductive predicate of Separation Logic. An important connection of that work with ours is the cutting of one edge on each cycle for modeling some cyclic structures by acyclic ones. Lee et al. define a *cut* rule, which removes one edge from each cycle and stores information about the edge in the grammar that corresponds to the data structure. Their mechanism is sufficiently precise to represent cyclic and panhandle lists. While Lee et al. do not discuss the application of their techniques to `Reverse`, their techniques should be capable of ensuring that the program has no unsafe memory operations or memory leaks when applied to a cyclic or a panhandle list. However, their analysis has no mechanism for relating the input and output data structures—a mechanism required for showing the total correctness of the algorithm.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments, as well as Hongseok Yang for important clarifications on the power and limitations of grammar-based shape analysis [9].

References

1. D Distefano, P O'Hearn, and H Yang. Interprocedural shape analysis with separated heap abstractions. In *Tools and Algs. for the Construction and Analysis of Systems*, pages 287–302, March 2006.

[11] In fact, the logic that we use is not decidable.

2. G. Dong and J. Su. Incremental maintenance of recursive views using relational calculus/SQL. *SIGMOD Record*, 29(1):44–51, 2000.
3. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis Symp.*, pages 240–260, August 2006.
4. W. Hesse. *Dynamic Computational Complexity*. PhD thesis, Dept. of Computer Science, University of Massachusetts, June 2003.
5. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive closure logics. In *Workshop on Computer Science Logic*, pages 160–174, September 2004.
6. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification via structure simulation. In *Computer-Aided Verification*, pages 281–294, July 2004.
7. N. Klarlund and M. Schwartzbach. Graph types. In *Symp. on Principles of Programming Languages*, January 1993.
8. S. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Symp. on Principles of Programming Languages*, pages 115–126, January 2006.
9. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *European Symp. On Programming*, pages 124–140, April 2005.
10. A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *Computer-Aided Verification*, pages 519–533, July 2005.
11. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Verification, Model Checking, and Abstract Interpretation*, pages 181–198, January 2005.
12. A. Møller and M. Schwartzbach. The pointer assertion logic engine. In *Conf. on Programming Language Design and Impl.*, pages 221–231, June 2001.
13. G. Nelson. Verifying reachability invariants of linked structures. In *Symp. on Principles of Programming Languages*, pages 38–47, January 1983.
14. T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symp. On Programming*, pages 380–398, April 2003.
15. J. Reynolds. Separation Logic: A logic for shared mutable data structures. In *Symp. on Logic in Computer Science*, pages 55–74, July 2002.
16. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Symp. on Principles of Programming Languages*, pages 296–309, January 2005.
17. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.
18. G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. To appear in *ACM Transactions on Computational Logic (TOCL)*.