Effectively-Propositional Reasoning about Reachability in Linked Data Structures [⋆]

Shachar Itzhaky[1], Anindya Banerjee[2], Neil Immerman[3], Aleksandar Nanevski[2], and Mooly Sagiv[1]

[1] Tel Aviv University, Tel Aviv, Israel
[2] IMDEA Software Institute, Madrid, Spain
[3] University of Massachusetts, Amherst, USA

Abstract. This paper proposes a novel method of harnessing existing SAT solvers to verify reachability properties of programs that manipulate linked-list data structures. Such properties are essential for proving program termination, correctness of data structure invariants, and other safety properties. Our solution is complete, i.e., a SAT solver produces a counterexample whenever a program does not satisfy its specification. This result is surprising since even first-order theorem provers usually cannot deal with reachability in a complete way, because doing so requires reasoning about transitive closure.

Our result is based on the following ideas: (1) Programmers must write assertions in a restricted logic without quantifier alternation or function symbols. (2) The correctness of many programs can be expressed in such restricted logics, although we explain the tradeoffs. (3) Recent results in descriptive complexity can be utilized to show that every program that manipulates potentially cyclic, singly- and doubly-linked lists and that is annotated with assertions written in this restricted logic, can be verified with a SAT solver.

We implemented a tool atop Z3 and used it to show the correctness of several linked list programs.

1 Introduction

This paper shows that it is possible to reason about reachability between dynamically allocated memory locations in potentially cyclic, singly-linked and doubly-linked lists using effectively-propositional reasoning. We present a novel method that can harness existing SAT solvers to verify reachability properties of programs that manipulate linked-list data structures, and to produce a concrete counterexample whenever a program does not satisfy its specification. This result is surprising because the natural specification of such programs involves quantifiers, inductive definitions and transitive closure, thus

[⋆] Itzhaky and Sagiv were funded by the European Research Council under the European Union's Seventh Framework Program (FP7/2007-2013) / ERC grant agreement no. [321174-VSSC] and by a grant from the Israel Science Foundation (652/11). Banerjee and Nanevski were partially supported by by Spanish MINECO projects TIN2009-14599-C03-02 Desafios, TIN2010-20639 Paran10, TIN2012-39391-C04-01 Strongsoft, EU NoE Project 256980 Nessos, AMAROUT grant PCOFUND-GA-2008-229599, and Ramon y Cajal grant RYC-2010-0743. Immerman was partially supported by NSF grant CCF 1115448.

precluding first-order, automatic theorem provers from dealing with reachability in a complete way.

$$\begin{tikzcd}
\boxed{state} \arrow[r, "\text{database-update}"] \arrow[d, "query"] & \boxed{state} \arrow[d, "query"] \\
\boxed{qstate} \arrow[r, "\text{query-update}"] & \boxed{qstate}
\end{tikzcd}$$

Fig. 1. The view update problem. Queries are expressed by formulas in a rich logic with transitive closure, but query-update is expressed essentially propositionally.

Two central observations underpin our method. (i) In programs that manipulate singly- and doubly-linked lists it is possible to express the 'next' pointer in terms of the reachability relation between list elements. This permits direct use of recent results in descriptive complexity [10]: we can maintain reachability with respect to heap mutation in a precise manner. Moreover, we can axiomatize reachability using quantifier free formulas. (ii) In order to handle statements which traverse the heap, we allow verification conditions (VCs) with $\forall^*\exists^*$ formulas so that they can be discharged by SAT solvers (as we explain shortly). However, we allow the programmer to only write assertions in a restricted fragment of FOL that disallows formulas with quantifier alternations but allows reflexive transitive closure. The main reason is that invariants occur both in the antecedent and in the consequent of the VC for loops; thus the assertion language has to be closed under negation.

The appeal to descriptive complexity stems from the fact that recently it has been applied to the view-update problem in databases. This problem has a pleasant parallel to the heap reachability update problem we are considering. In the view-update problem, the logical complexity of updating a query wrt. database modifications is lower than computing the query for the updated database from scratch (depicted in Fig. 1). Indeed, the latter uses formulas with transitive closure, while the former uses quantifier-free formulas without transitive closure. In our setting, we compute reachability relations instead of queries. We exploit the fact that the logical complexity of adapting the (old) reachability relation to the updated heap is lower than computing the new reachability relation from scratch. The solution we employ is similar to the use of dynamic graph algorithms for solving the view-update problem, where directed paths between nodes are updated when edges are added/removed (e.g., see [5]), except that our solution is geared towards verification of heap-manipulating programs with linked data structures.

Main Results

– We define AF^R, a new logic for expressing properties of programs, that is an *alternation free* sub-fragment of FO^{TC} (i.e., first-order logic with transitive closure): alternation between universal and existential quantifiers in formulas is disallowed. A distinguishing feature of AF^R is that it allows relation symbols but does not allow direct application of function symbols. Atomic formulas of AF^R may denote reachability relations between memory locations via pointers such as $next$ and $prev$ fields in linked lists, or any other relations without transitive closure.
– We empirically show that loop invariants in many programs manipulating singly- and doubly-linked lists can be specified using AF^R formulas.

– We show that the effect of many procedures manipulating singly- and doubly-linked lists can be specified using AF^R formulas. This result may require that the memory that the procedure manipulates be “owned” by its formal parameters.
– We show direct use of existing results in dynamic complexity [10] to prove that AF^R formulas are closed under weakest preconditions for statements which destructively update memory (e.g., $x.next := y$).
– For statements that traverse the heap (e.g., $x := y.next$), AF^R formulas are *not* closed under weakest preconditions. For these cases we show that weakest preconditions are expressible in the AE^R logic which generalizes AF^R by permitting existential quantification inside universal quantification. AE^R formulas are decidable for validity since their negation has the form $\exists^*\forall^*$, and fits in the Bernays-Schönfinkel fragment which is decidable for satisfiability [19]. In fact, they can be checked with a SAT solver by replacing existential quantifiers with constants, and universal quantifications by conjunctions over the constants. Indeed, Z3 [4] is complete for these formulas.
– We report on experiments with a tool that checks correctness of several, commonly used heap-manipulating structured programs, and that uses Z3 as back-end. The tool can determine whether or not program annotations (pre- and postconditions, loop invariants) are AF^R formulas, and can check both safety and equivalence of procedures. The tool is sound and also complete in the sense that it generates concrete counterexamples for programs violating the VCs.

This paper is accompanied by a technical report containing further examples and proofs.

2 Overview

2.1 Programming with Restricted Invariants

In this paper we require that the specified invariants are AF^R formulas. That is, they only use reflexive transitive closure but do not explicitly use function symbols and quantifier alternations.

Definition 1. *Let $t_1, t_2, \dots t_n$ be logical variables or constant symbols. We define four types of* ***atomic propositions****: (i) $t_1 = t_2$ denoting equality, (ii) $r(t_1, t_2, \dots, t_n)$ denoting the application of relation symbol r of arity n, and (iii) $t_1\langle f^*\rangle t_2$ denoting the existence of $k \geq 0$ such that $f^k(t_1) = t_2$, where $f^0(t_1) \overset{\text{def}}{=} t_1$, and $f^{k+1}(t_1) \overset{\text{def}}{=} f(f^k(t_1))$. We say that $t_1\langle f^*\rangle t_2$ is a* ***reachability constraint*** *between t_1 and t_2 via the function f.* ***Quantifier-free formulas*** *(QF^R) are Boolean combinations of such formulas without quantifiers.* ***Alternation-free formulas*** *(AF^R) are Boolean combinations of such formulas with additional quantifiers of the form $\forall^*{:}\varphi$ or $\exists^*{:}\varphi$ where φ is a QF^R formula.* ***Forall-Exists Formulas*** *(AE^R) formulas are Boolean combinations of such formulas with additional quantifiers of the form $\forall^*\exists^*{:}\varphi$ where φ is a QF^R formula. In particular, $QF^R \subset AF^R \subset AE^R$.*

Fig. 2 presents a Java program for in-situ reversal of a linked list. Every node of the list has a *next* field that points to its successor node in the list. Thus, we can model *next* as a function that maps a node in the list to its successor. For simplicity we assume that the program manipulates the entire heap, that is, the heap consists of just the nodes in the linked list. To describe the heap that is reachable from the formal parameter h, where h points to the head of the input list, we use the formula $\forall\alpha : h\langle \mathit{next}^*\rangle\alpha$.

We also assume, until Section 5, that the heap is acyclic, i.e., the formula ac below is a precondition of $reverse$.

$$ac \overset{\text{def}}{=} \forall\alpha,\beta : \alpha\langle \mathit{next}^*\rangle\beta \wedge \beta\langle \mathit{next}^*\rangle\alpha \rightarrow \alpha = \beta \tag{1}$$

$$\boxed{\begin{array}{c}
I_0 \overset{\text{def}}{=} ac \wedge \forall\alpha : h\langle \mathit{next}^*\rangle\alpha \\
I_3 \overset{\text{def}}{=} ac \wedge \forall\alpha,\beta \neq \mathit{null} : \left\{ \begin{array}{ll} \alpha\langle \mathit{next}^*\rangle\beta \Leftrightarrow \beta\langle \mathit{next}_0^*\rangle\alpha & d\langle \mathit{next}^*\rangle\alpha \\ c\langle \mathit{next}^*\rangle\alpha \wedge (\alpha\langle \mathit{next}^*\rangle\beta \Leftrightarrow \alpha\langle \mathit{next}_0^*\rangle\beta) & \neg d\langle \mathit{next}^*\rangle\alpha \end{array} \right\} \\
I_9 = ac \wedge \forall\alpha : d\langle \mathit{next}^*\rangle\alpha \wedge (\forall\alpha,\beta : \alpha\langle \mathit{next}^*\rangle\beta \Leftrightarrow \beta\langle \mathit{next}_0^*\rangle\alpha)
\end{array}}$$

Table 1. AF^R invariants for $reverse$. Note that $\mathit{next}, \mathit{next}_0$ are function symbols while $\alpha\langle \mathit{next}^*\rangle\beta$, $\alpha\langle \mathit{next}_0^*\rangle\beta$ are atomic propositions on the reachability via directed paths from α to β consisting of next, next_0 edges.

```
Node reverse(Node h) {
  0: Node c = h;
  1: Node d = null;
  2: while  3: (c != null) {
       4: Node t = c.next;
       5: c.next = null;
       6: c.next = d;
       7: d = c;
       8: c = t;
  }
  9: return d;
}
```

Fig. 2. A simple Java program that reverses a list in-situ.

Table 1 shows the invariants I_0, I_3 and I_9 that describe a precondition, a loop invariant, and a postcondition of $reverse$. They are expressed in AF^R which permits use of function symbols (e.g. *next*) in formulas only to express reachability (cf. next^*); moreover, quantifier alternation is not permitted. The notation $\left\{ \begin{array}{ll} f & b \\ g & \neg b \end{array} \right\}$ is shorthand for the conditional $(b \wedge f) \vee (\neg b \wedge g)$.

Note that I_3 and I_9 refer to next_0, the value of *next* at procedure entry. The postcondition I_9 says that $reverse$ preserves acyclicity of the list and updates *next* so that, upon procedure termination, the links of the original list have been reversed. It also says that all the nodes are reachable from d in the reversed list. I_3 says that at loop entry c is non-null and moreover, the original list is partially reversed. That is, any node reachable from d is connected in reverse wrt. the input list, whereas any node not reachable from d is reachable from c and belongs to the part of the list that has not yet been reversed. Observe that I_3 and I_9 only refer to next^* and never to *next* alone. A more natural way to express I_9 would be

$$I_9' \overset{\text{def}}{=} ac \wedge \forall\alpha : d\langle \mathit{next}^*\rangle\alpha \wedge (\forall\alpha,\beta : \mathit{next}(\alpha) = \beta \Leftrightarrow \mathit{next}_0(\beta) = \alpha) \tag{2}$$

But this formula is not in AF^R because it explicitly refers to function symbols *next* and next_0 outside a reachability constraint.

2.2 Inverting Reachability Constraints

A crucial step in moving from arbitrary $\mathit{FO}^{\mathit{TC}}$ formulas to AF^R formulas is eliminating explicit uses of functions such as *next*. While this may be difficult for a general graph, we show that this can be done for programs that manipulate (potentially cyclic) singly- and doubly-linked lists. In this section, we informally demonstrate this elimination for acyclic lists. We observe that if *next* is acyclic, we can construct next^+ from next^* by

$$\alpha\langle \mathit{next}^+\rangle\beta \Leftrightarrow \alpha\langle \mathit{next}^*\rangle\beta \wedge \alpha \neq \beta \qquad (3)$$

Also, since *next* is a function, the set of nodes reachable from a node α is totally ordered by next^*. Therefore, $\mathit{next}(\alpha)$ is the minimal node in this order that is not α. The minimality is expressed using extra universal quantification in

$$\mathit{next}(\alpha) = \beta \Leftrightarrow \alpha\langle \mathit{next}^+\rangle\beta \wedge \forall\gamma : \alpha\langle \mathit{next}^+\rangle\gamma \rightarrow \beta\langle \mathit{next}^*\rangle\gamma \qquad (4)$$

This inversion shows that *next* can be expressed using AF^R formulas. However, caution must be practiced when using the elimination above, because it may introduce alternations (see [2]). Nevertheless our experiments demonstrate that in a number of commonly occurring examples, the alternation can be removed or otherwise avoided, yielding an equivalent AF^R formula.

2.3 Generating AE^R Verification Conditions

Given a program annotated with loop invariants and procedure specifications, it is possible to automatically generate VCs to check that the invariants are satisfied by all program executions (e.g., see [8]). For example, the VC of $\mathit{reverse}$ asserts that every execution which starts in a state satisfying I_0 satisfies I_3 and that I_3 is indeed *inductive*. That is, if it holds on the loop entry and if the loop is executed, I_3 remains true after the execution. Finally, the VC asserts that I_3 and the negation of the loop condition implies the postcondition I_9.

For simplicity, we do not handle deallocation operations here. Since our logic expresses reachability it does not depend on a particular memory abstraction, and can handle both garbage collection and programs with explicit deallocation.

Unfortunately showing validity of formulas with transitive closure and quantifier alternations, i.e., nesting existential inside universal quantifiers or vice versa is very difficult for first-order theorem provers: existing decision procedures cannot handle such formulas, because even the simplest use of transitive closure leads to undecidability [11].

In this paper we show that for programs with AF^R assertions manipulating singly- and doubly-linked lists, the generated VCs are effectively propositional. However, AF^R formulas are not powerful enough to describe the VCs of programs with AF^R invariants. The main reason is that the semantics of accessing heap fields, e.g., $x := y.next$ requires one level of alternation. Therefore, we slightly generalize AF^R and generate VCs that have the form $\forall^*\exists^* : \varphi$ where φ is a quantifier-free formula which does not contain function symbols in terms but may contain reachability and relation symbols. Validity of formulas in this class, AE^R, are decidable since their negations have the form $\exists^*\forall^*{:}\varphi$, that is, they belong to the Bernays-Schönfinkel class of formulas [19]. In fact, the formulas can be checked with a SAT solver by replacing existential quantifiers with distinct Skolem constants, and then grounding all universally quantified variables by all combinations of constants. Indeed, Z3 handles these formulas in a precise manner without the need to perform this transformation.

We show that AE^R formulas are closed under weakest preconditions (*wp*), i.e., for every statement S and postcondition Q expressed as AE^R formula, it must be the case that $wp(S, Q)$ is expressed as an AE^R formula. To show this closure property of AE^R formulas, we rely on recent results in descriptive complexity which prove that for singly-linked data structures edge mutations are expressible *without* quantifications [10]. Specifically, this means that updates to the reachability relation, wrt. pointer removals and additions, can be expressed using quantifier-free formulas. We note, however, that our applications to program verification go beyond descriptive complexity in several major ways: (i) Programs can create fresh nodes as a result of dynamic allocation statements of the form $x := \mathsf{new}$. (ii) A heap field read, $x := y.next$, does not mutate the heap but can affect the truth value of reachability constraints. (iii) Calls to libraries can mutate the heap in an unbounded way. (iv) In order to guarantee correctness of loops and procedures, the verification is conducted modularly using AF^R invariants, pre- and postconditions. For example, to verify the correctness of a code which includes a procedure call, we assert that the states at the call satisfy the procedure's precondition expressed as an AF^R formula and assert that after the call the state satisfies the procedure's postcondition specified by an AF^R formula.

Handling Destructive Updates. We first handle the case of statements that assign null to pointer fields and so remove directed paths. For example, statement 5 in the $reverse$ program is modeled by

$$wp(c.next := \mathsf{null}, Q) \overset{\text{def}}{=} \begin{array}{l} c \neq null \\ \wedge\ Q[\alpha\langle next^*\rangle\beta \wedge (\neg\alpha\langle next^*\rangle c \vee \beta\langle next^*\rangle c)/\alpha\langle next^*\rangle\beta] \end{array} \tag{5}$$

The assignment removes the outgoing edge from the node pointed to by c. This is a simplified condition that also uses the fact that the manipulated list is acyclic. An operation of the form $c.next := \mathsf{null}$ deletes an existing path between nodes α and β if the path goes through a (non-null) node c. This situation can be expressed by the formula $\alpha\langle next^*\rangle c \wedge \neg\beta\langle next^*\rangle c$. So the negation of this formula conjoined with $\alpha\langle next^*\rangle\beta$ must hold in the precondition so that $\alpha\langle next^*\rangle\beta$ holds in the postcondition. Notice that this rule drastically differs from the standard McCarthy axiom [16], which directly assigns a new value to the heap:

$$wp'(c.next := \mathsf{null}, Q) \overset{\text{def}}{=} Q[next[c \mapsto null]/next]$$

We forbid the use of this rule for it uses a function (*next*) and relies on "recomputing" reachability constraints in Q by using the transitive closure of $next[c \mapsto null]$. Instead, we directly update the effect on the reachability relation $\alpha\langle next^*\rangle\beta$ by substituting it with a quantifier-free formula shown in (5). A similar definition exists for *wp* for statements like $c.next := d$ that add edges, as we show later in Table 3.

Surprisingly, the semantics of field dereference statement $t := c.\mathit{next}$ is a bit more subtle despite the fact that such a statement does not modify the heap. However, a *wp* for field dereference can also be given in AE^R (see Section 3), thus enabling verification with a SAT solver in a complete way.

As shown by Hesse [10], a QF^R definition of the effect on reachability can be also done for cyclic data structures with a single pointer field. However, for programs with reachability over more than one field in general DAGs, quantifiers are required [6].

2.4 Decidability of AE^R

Reachability constraints written as $\alpha\langle \mathit{next}^*\rangle\beta$ are not directly expressible in FOL. However, AE^R formulas can be reduced to first-order $\forall^*\exists^*$ formulas without function symbols (which are decidable; see Section 2.3) in the following fashion: Introduce a new binary relation symbol $\widehat{n^*}$ with the intended meaning that $\widehat{n^*}(\alpha, \beta) \Leftrightarrow \alpha\langle \mathit{next}^*\rangle\beta$. Even though $\widehat{n^*}$ is an uninterpreted relation, we will consistently maintain the fact that it models reachability. Every formula φ is translated into

$$\varphi' \overset{\text{def}}{=} \varphi[\widehat{n^*}(t_1, t_2)/t_1\langle \mathit{next}^*\rangle t_2]$$

For example, the acyclicity relation shown in (1) is translated into:

$$\widehat{ac} \overset{\text{def}}{=} \forall\alpha, \beta : \widehat{n^*}(\alpha, \beta) \land \widehat{n^*}(\beta, \alpha) \to \alpha = \beta \tag{6}$$

We add the consistency rule Γ_{linOrd} shown in Table 2, which requires that $\widehat{n^*}$ is a total order. In Section 3 and in [2] we prove that the translated formula $\Gamma_{\text{linOrd}} \to \varphi'$ is valid if and only if the original formula φ is valid. The proof constructs real models from "simulated" FO models using the reachability inversion (4).

$$\boxed{\begin{aligned}\Gamma_{\text{linOrd}} \overset{\text{def}}{=} &\forall\alpha, \beta : \widehat{n^*}(\alpha, \beta) \land \widehat{n^*}(\beta, \alpha) \leftrightarrow \alpha = \beta \quad \land \\ &\forall\alpha, \beta, \gamma : \widehat{n^*}(\alpha, \beta) \land \widehat{n^*}(\beta, \gamma) \to \widehat{n^*}(\alpha, \gamma) \quad \land \\ &\forall\alpha, \beta, \gamma : \widehat{n^*}(\alpha, \beta) \land \widehat{n^*}(\alpha, \gamma) \to (\widehat{n^*}(\beta, \gamma) \lor \widehat{n^*}(\gamma, \beta))\end{aligned}}$$

Table 2. Γ_{linOrd} says all points reachable from a given point are linearly ordered.

2.5 Expressivity of AF^R

Although AF^R is a relatively weak logic, it can express interesting properties of lists. Typical predicates that express disjointness of two lists and sharing of tails are expressible in AF^R. For example, for two singly-linked lists with headers h, k, $\mathit{disjoint}(h, k) \Leftrightarrow \forall\alpha : \alpha \neq \mathit{null} \to \neg(h\langle \mathit{next}^*\rangle\alpha \land k\langle \mathit{next}^*\rangle\alpha)$.

Another capability still within the power of AF^R is to relax the earlier assumption that the program manipulates the whole memory. We describe a summary of $\mathit{reverse}$ on arbitrary acyclic linked lists in a heap that may contain other linked data structures. Realistic programs obey ownership requirements, e.g., the head h of the list *owns* the input list which means that it is impossible to reach one of the list nodes without passing through h. That is,

$$\forall\alpha, \beta : \alpha \neq \mathit{null} \to (h\langle \mathit{next}^*\rangle\alpha \land \beta\langle \mathit{next}^*\rangle\alpha) \to h\langle \mathit{next}^*\rangle\beta \tag{7}$$

This requirement is conjoined to the precondition, ac, of $reverse$. Its postcondition is the conjunction of ac, the fact that h_0 and d reach the same nodes, (i.e., $\forall\alpha : h_0\langle next_0^*\rangle\alpha \Leftrightarrow d\langle next^*\rangle\alpha$) and

$$\forall\alpha,\beta : \alpha\langle next^*\rangle\beta \Leftrightarrow \left\{\begin{array}{ll} \beta\langle next_0^*\rangle\alpha \wedge \beta \neq null & h_0\langle next_0^*\rangle\alpha \wedge h_0\langle next_0^*\rangle\beta \\ \alpha\langle next_0^*\rangle\beta & \neg h_0\langle next_0^*\rangle\alpha \wedge \neg h_0\langle next_0^*\rangle\beta \\ \mathbf{false} & h_0\langle next_0^*\rangle\alpha \wedge \neg h_0\langle next_0^*\rangle\beta \\ \alpha\langle next_0^*\rangle h_0 \wedge \beta = h_0 & \neg h_0\langle next_0^*\rangle\alpha \wedge h_0\langle next_0^*\rangle\beta \end{array}\right\} \quad (8)$$

Here, the bracketed formula should be read as a four-way case, i.e., as disjunction of the formulas $h_0\langle next_0^*\rangle\alpha \wedge h_0\langle next_0^*\rangle\beta \wedge \beta\langle next_0^*\rangle\alpha \wedge \beta \neq null$; $\neg h_0\langle next_0^*\rangle\alpha \wedge \neg h_0\langle next_0^*\rangle\beta \wedge \alpha\langle next_0^*\rangle\beta$; $h_0\langle next_0^*\rangle\alpha \wedge \neg h_0\langle next_0^*\rangle\beta \wedge \mathbf{false}$; and, $\neg h_0\langle next_0^*\rangle\alpha \wedge h_0\langle next_0^*\rangle\beta \wedge \alpha\langle next_0^*\rangle h_0 \wedge \beta = h_0$. Intuitively, this summary distinguishes between the following four cases: (i) both the source (α) and the target (β) are in the reversed list (ii) both source and target are outside of the reversed list (iii) the source is in the reversed list and the target is not, and (iv) the source is outside and the target is in the reversed list. Cases (i)–(iii) are self-explanatory. For (iv) reachability can occur when there exists a path from α to $h_0 = \beta$. Formula (8) is in AF^R. In terms of [21], this means that we assume that the procedure is cutpoint free. We can also generate an AF^R summary for a program with *fixed* number of cutpoints, as is done in Section 5.

The general case of unbounded number of cutpoints requires a formula that is outside AF^R. A non-AF^R formula also arises when we want to express that a program manipulates two lists of equal length; such a formula requires an inductive definition. See [2] for examples of these formulas.

3 Weakest Preconditions of Atomic Heap Manipulating Statements

In this section we show how to express the weakest liberal preconditions of atomic heap manipulating statements using AE^R formulas, for programs that manipulate acyclic singly-linked lists. Table 3 shows standard *wp* computation rules (top part) and the corresponding rules for field update, field read and dynamic allocation (bottom part). The correctness of the rule for destructive field update is according to Hesse's thesis [10].

Field Dereference. The rationale behind the formula for $wp(x := y.next, Q)$ is that if y has a successor, then the formula Q should be satisfied when x is replaced by this successor. The natural way to specify this is using the Hoare assignment rule

$$wp'(x := y.next, Q) \stackrel{\text{def}}{=} Q[next(y)/x]$$

However, this rule uses the function *next* and does not directly express reachability. Instead we will construct a relation r_{next} such that $r_{next}(\alpha,\beta) \Leftrightarrow next(\alpha) = \beta$ and then use universal quantifications to "access" the value

$$wp''(x := y.next, Q) \stackrel{\text{def}}{=} \forall\alpha : r_{next}(y,\alpha) \rightarrow Q[\alpha/x]$$

$$\begin{aligned}
wp(\mathsf{skip}, Q) &\overset{\text{def}}{=} Q\\
wp(x := y, Q) &\overset{\text{def}}{=} Q[y/x]\\
wp(S_1\ ;\ S_2, Q) &\overset{\text{def}}{=} wp(S_1, wp(S_2, Q))\\
wp(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, Q) &\overset{\text{def}}{=} [\![B]\!] \wedge wp(S_1, Q) \vee\\
&\quad\ \neg[\![B]\!] \wedge wp(S_2, Q)\\
wp(\mathsf{while}\ B\ \{I\}\ \mathsf{do}\ S, Q) &\overset{\text{def}}{=} I
\end{aligned}$$

$$\begin{aligned}
wp(x.\mathit{next} := \mathsf{null}, Q) &\overset{\text{def}}{=} Q[\alpha\langle \mathit{next}^*\rangle\beta \wedge (\neg\alpha\langle \mathit{next}^*\rangle x \vee \beta\langle \mathit{next}^*\rangle x)/\alpha\langle \mathit{next}^*\rangle\beta]\\
wp(x.\mathit{next} := y, Q) &\overset{\text{def}}{=} \neg y\langle \mathit{next}^*\rangle x \wedge\\
&\quad\ Q[\alpha\langle \mathit{next}^*\rangle\beta \vee (\alpha\langle \mathit{next}^*\rangle x \wedge y\langle \mathit{next}^*\rangle\beta)/\alpha\langle \mathit{next}^*\rangle\beta]\\
wp(x := \mathsf{new}, Q) &\overset{\text{def}}{=} \forall\alpha : \left(\bigwedge\nolimits_{p\in \mathit{Pvar}\cup\{\mathit{null}\}} \neg p\langle \mathit{next}^*\rangle\alpha\right) \to Q[\alpha/x]\\
P_{\mathit{next}^+} &\overset{\text{def}}{=} s\langle \mathit{next}^*\rangle t \wedge s \neq t\\
P_{\mathit{next}} &\overset{\text{def}}{=} P_{\mathit{next}^+} \wedge \forall\gamma : P_{\mathit{next}^+}[\gamma/t] \to \gamma\langle \mathit{next}^*\rangle t\\
wp(x := y.\mathit{next}, Q) &\overset{\text{def}}{=} \forall\alpha : P_{\mathit{next}}[y/s, \alpha/t] \to Q[\alpha/x]
\end{aligned}$$

Table 3. Rules for computing weakest liberal preconditions for procedures annotated with loop invariants and postconditions. I denotes the loop invariant, $[\![B]\!]$ is the AF^R formula for program conditions and Q is the postcondition expressed as an AF^R formula. The top frame shows the standard *wp* rules for While-language, the bottom frame contains our additions for heap updates, memory allocation, and dereference.

Since *next* is acyclic, we can express r_{next} in terms of next^* as follows. First we observe that $\mathit{next}(\alpha) \neq \alpha$. Also, since *next* is a function, the set of nodes reachable from α is totally ordered by next^*. Therefore, similarly to Section 2.2, we can express $r_{\mathit{next}}(\alpha, \beta)$ as the minimal node β in this order where $\beta \neq \alpha$. Expressing minimality "costs" one extra universal quantification.

In Table 3, formula P_{next} expresses r_{next} in terms of next^*: P_{next} holds if and only if there is a path of length 1 between s and t (source and target). Thus, $P_{\mathit{next}}[y/s, \alpha/t]$ is satisfied exactly when $\alpha = \mathit{next}(y)$. If y does not have a successor, then $P_{\mathit{next}}[y/s, \alpha/t]$ can only be $\mathbf{true}$ if $\alpha = \mathit{null}$, hence Q should be satisfied when x is replaced by *null*, which is in line with the concrete semantics. A central lemma in [2] shows that the formula P_{next} correctly defines *next* as a relation.

Dynamic allocation. The rule $wp(x := \mathsf{new}, Q)$ expresses the semantic uncertainty caused by the behavior of the memory allocator. We want to be compatible with any run-time memory management, so we do not enforce a concrete allocation policy, but require that the allocated node meets some reasonable specifications, namely, that it is different from all values stored in program variables, and that it is unreachable from any other node allocated previously (Note: for programs with explicit `free()`, this assumption relies on the absence of dangling pointers, which can be verified by introducing appropriate assertions; this is, however, beyond the scope of this paper).

4 Generating an AE^R Verification Condition

Table 4 provides the standard rules for computing VCs using weakest liberal preconditions. An auxiliary function VC_{aux} is used for defining the set of side conditions for the loops occurring in the program. These rules are standard and their soundness and relative completeness have been discussed elsewhere (e.g. see [8]).

We assume that the effect, $[\![B]\!]$, of the condition B used in the conditional and the while loop, is defined by an AF^R formula. We also assume that all loop invariants I, the precondition P, and postcondition Q are AF^R formulas. The rule for while loop is split into two parts: in the *wp* we take just the loop invariant, where VC_{aux} asserts that loop invariants are inductive and implies the postcondition for each loop.

The rules may generate exponential formulas. Another solution can be implemented either using the method of Flanagan and Saxe [7] or by using a set of symbols for every program point.

$$\begin{array}{|c|}\hline VC_{aux}(S, Q) \overset{\text{def}}{=} \varnothing \qquad \text{(for any atomic command } S\text{)} \\ VC_{aux}(S_1; S_2, Q) \overset{\text{def}}{=} VC_{aux}(S_1, wp(S_2, Q)) \cup VC_{aux}(S_2, Q) \\ VC_{aux}(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, Q) \overset{\text{def}}{=} VC_{aux}(S_1, Q) \cup VC_{aux}(S_2, Q) \\ VC_{aux}(\mathsf{while}\ B\ \{I\}\ \mathsf{do}\ S, Q) \overset{\text{def}}{=} VC_{aux}(S, I) \ \cup \\ \qquad\qquad \{I \land [\![B]\!] \to wp(S, I), I \land \neg[\![B]\!] \to Q\} \\ \hline VC_{gen}(\{P\}S\{Q\}) \overset{\text{def}}{=} P \to wp(S, Q) \land \bigwedge VC_{aux}(S, Q) \\ \hline \end{array}$$

Table 4. Standard rules for computing VCs using weakest liberal preconditions for procedures annotated with loop invariants and pre/postconditions.

Notice that Table 4 only uses weakest liberal preconditions in a positive context without negations. Therefore, the following proposition (proof in [2]) holds.

Proposition 1 (VCs in AE^R). *For every program S whose precondition P, postcondition Q, branch conditions, loop conditions, and loop invariants are all expressed as AF^R formulas, $VC_{gen}(\{P\}S\{Q\})$ is in AE^R.*

Optimization remark. The size of the VC can be significantly reduced if instead of syntactic substitution, we introduce a new vocabulary for each substituted atomic formula, axiomatizing its meaning as a separate formula. For example, $Q[P(\alpha, \beta)/\alpha\langle next^*\rangle\beta]$ (where P is some formula with free variables α, β), can be written more compactly as $Q[r_1(\alpha, \beta)/\alpha\langle next^*\rangle\beta] \land \forall\alpha, \beta : r_1(\alpha, \beta) \Leftrightarrow P(\alpha, \beta)$, where r_1 is a fresh relational symbol. When Q contains many applications of $\langle next^*\rangle$ and P is large, this may save a lot of formula space; roughly, it reduces the order of the VC size from quadratic to linear. Our original implementation employed this optimization, which is also nice for finding bugs — when the program violates the invariants the SAT solver produces a counterexample with the concrete states at every program point. The approach of [7] is also applicable in this case.

5 Extensions

Doubly-linked List and Nested Lists. To verify a program that manipulates a doubly-linked list, all that needs to be done is to duplicate the analysis we did for *next*, for a second pointer field *prev*. As long as the only atomic formulas used in assertions are $\alpha\langle next^*\rangle\beta$ and $\alpha\langle prev^*\rangle\beta$ (and not, for example, $\alpha\langle(next|prev)^*\rangle\beta$), providing the substitutions for atomic formulas in Table 3 would not get us outside of the class AE^R. In particular, we have verified the doubly-linked list property:

$$\forall\alpha,\beta : h\langle next^*\rangle\alpha \wedge h\langle next^*\rangle\beta \rightarrow (\alpha\langle next^*\rangle\beta \Leftrightarrow \beta\langle prev^*\rangle\alpha).$$

In fact we can verify nested lists and, in general, lists with arbitrary number of pointer fields as long as reachability constraints are expressed using only one function symbol at a time, like in the case of $next$ and $prev$ above.

Cycles. For data structures with a single pointer, the acyclicity restriction may be lifted by using an alternative formulation that keeps and maintains more auxiliary information [10, 13]. Instead of keeping track of just $next^*$, we instrument the edge addition operation with a check: if the added edge is about to close a cycle, then instead of adding the edge, we keep it in a separate set M of "cycle-inducing" edges. Two properties of lists now come into play: (1) The number of cycles reachable from program variables, and hence the size of M, is bounded by the number of program variables; (2) Any path (simple or otherwise) in the heap may utilize at most one of those edges, because once a path enters a cycle, there is no way out. In all assertions, therefore, we replace $\alpha\langle next^*\rangle\beta$ with: $\alpha\langle next^*\rangle\beta \vee \bigvee_{\langle u,v\rangle\in M}(\alpha\langle next^*\rangle u \wedge v\langle next^*\rangle\beta)$. Notice that it is possible to construct this formula thanks to the bound on the size of M; otherwise, an existential quantifier would have been required in place of the disjunction.

More details and cases for cycles can be found in the Technical Report [2].

Bounded Sharing. Arbitrary sharing in data structures is hard, because even in lists, any node of the list may be shared (that is, have more than one incoming edge). In this case we have to use quantification since we do not know in advance which node in the list is going to be a cutpoint for which other nodes. However, when the *entire* heap consists solely of lists, the quantifier may be replaced with a disjunction if we take into account that there is a bounded number of program variables, which can serve as the heads of lists, and any two lists have at most one cutpoint. Such heaps when viewed as graphs are much simpler than general DAGs, since one can define in advance a set of *constant symbols* to hold the edges that induce the sharing; for example, if we have one list through the nodes $x \rightarrow u_1 \rightarrow u_2$ and a second list through $y \rightarrow v_1 \rightarrow v_2$, all distinct locations, then adding an edge $u_2 \rightarrow v_1$ would create sharing, as the nodes v_1, v_2 become accessible from both x and y. This technique is also covered by Hesse [10].

6 Composing Procedure Summaries to Check Program Equivalence

This section argues that AF^R-postconditions of procedure summaries can be sequentially composed and used to check if two pieces of code are equivalent, i.e., that they produce the same output for a given input.

Illustrating $reverse(reverse\ h) = h$. Let next_1^* denote the reachability after running the inner $reverse$, and let next_2^* denote the reachability after running the outer $reverse$. We can express the equivalence of $reverse(reverse\ h)$ and h using the following AF^R implication:

$$\begin{aligned}(\forall\alpha,\beta : \alpha\langle \mathit{next}_1^*\rangle\beta \Leftrightarrow \beta\langle \mathit{next}_0^*\rangle\alpha) \land (\forall\alpha,\beta : \alpha\langle \mathit{next}_2^*\rangle\beta \Leftrightarrow \beta\langle \mathit{next}_1^*\rangle\alpha)\\ \to \forall\alpha,\beta : \alpha\langle \mathit{next}_2^*\rangle\beta \Leftrightarrow \alpha\langle \mathit{next}_0^*\rangle\beta\end{aligned} \tag{9}$$

The second conjunct of the implication's antecedent describes the effect of the inner $reverse$ on the initial state while the third conjunct describes the effect of the outer $reverse$ on the state resulting from the first. The consequent of the implication states that the initial and final states are equivalent.

Illustrating $\mathit{filter}(C, reverse(h)) = reverse(\mathit{filter}(C, h))$. The program *filter* takes a unary predicate C on nodes, and a list with head h, and returns a list with all nodes satisfying C removed. The postcondition of *filter* is: $\forall\alpha,\beta : \alpha\langle \mathit{next}^*\rangle\beta \Leftrightarrow \neg C(\alpha) \land \neg C(\beta) \land \alpha\langle \mathit{next}_0^*\rangle\beta$. It says that β is reachable from α in the filtered list provided neither α nor β satisfies C and β was reachable from α initially. We show ([2]) that the equivalence of $\mathit{filter}(C, reverse(h))$ and $reverse(\mathit{filter}(C, h))$ can be expressed using an AF^R implication.

7 Experimental Results

7.1 Details

We have implemented a VC generator, according to Tables 3 and 4, in Python, and PLY (Python Lex-Yacc) is employed at the front-end to parse While-language programs annotated with AF^R assertions. The tool verifies that invariants are in the class AF^R and have reachability constraints along a single field (of the form f^*). The assertions may refer to the store and heap at the entry to the procedure via x_0, f_0, etc. SMT-LIB v2 [1] standard notation is used to format the VC and to invoke Z3. The validity of the VC can be checked by providing its negation to Z3. If Z3 exhibits a satisfying assignment then that serves as counterexample for the correctness of the assertions. If no satisfying assignment exists, then the generated VC is valid, and therefore the program satisfies the assertions.

The output model/counterexample (S-Expression), if one is generated, is then also parsed, so that we have the truth table of *next**. This structure represents the state of the program either at entry or at the beginning of a loop iteration: running the program from this point will violate one or more invariants. To provide feedback to the user, *next* is recovered by computing (4), and then the `pygraphviz` tool is used to visualize and present to the user a directed graph, whose vertices are nodes in the heap, and whose edges are the *next* pointer fields.

We also implemented two procedures for generating VCs: the first one implements the standard rules shown in Table 4 and a second one uses a separate set of relation and constant symbols per program point as a way to reduce the size of the generated VC formula. We only report data on the former since it exhibited better running times.

7.2 Verification Examples

We have written AF^R loop invariants and procedure pre- and postconditions for 13 example procedures shown in Table 6. These are standard benchmarks and what they do can be inferred either from their names or from Table 5. We are encouraged by the fact that it was not difficult to express assertions in AF^R for these procedures. The annotated examples and the VC generation tool are publicly available from `http://www.cs.tau.ac.il/~shachar/afwp.html`.

For an example of the annotations used in the benchmarks, see Table 1, listing the precondition, loop invariant, and postcondition of *reverse*.

As expected, Z3 is able to verify all the correct programs. Table 6 shows statistics for size and complexity of the invariants and the running times for Z3.

To give some account of the programs' sizes, we observe the program summary specification given as pre- and postcondition, count the number of atomic formulas in each of them, and note the depth of quantifier nesting; all our samples had only universal quantifiers. We did the same for each program's loop invariant and for the generated VC_{gen}. Naturally, the size of the VC grows rapidly —approximately at a quadratic rate. This can be observed in the result of the measurements for "SLL: merge", where (i) the size of the invariant and (ii) the number of if-branches and heap manipulating statements, was larger than those in other examples. Still, the time required by Z3 to prove that the VC is valid is short.

For comparison, the size of the formula generated by the alternative implementation, using a separate set of symbols for each program location, was about 10 times shorter — 239 atomic formulas. However, Z3 took a significantly longer time, at 1357ms. We therefore preferred to use the first implementation.

Thanks to the fact that FOL-based tools, and in particular SAT solvers, permit multiple relation symbols we were able to express ordering properties in sorted lists, and so verify order-aware programs such as "insert" and "merge". This situation can be contrasted with tools like Mona ([12],[9]) which are based on monadic second-order logic, where multiple relation symbols are disallowed.

Additionally, we made experiments in composing summaries of *filter* and *reverse* (Section 6). In this case, we wrote the formulas manually and ran Z3 on them, to get a proof of the validity of the equivalences.

SLL: insert	— Adds a node into a sorted list, preserving order.
SLL: find	— Locates the first item in the list with a given value.
SLL: last	— Returns the last node of the list.
SLL: merge	— Merges two sorted lists into one, preserving order.
SLL: swap	— Exchanges the first and second element of a list.
DLL: fix	— Directs the back-pointer of each node towards the previous node, as required by data structure invariants.
DLL: splice	— Splits a list into two well-formed doubly-linked lists.

Table 5. Description of some linked list manipulating programs verified by our tool.

Table 6. Implementation Benchmarks; P,Q — program's specification given as pre- and post-condition, I — loop invariant, VC — verification condition, # — number of atomic formulas, $\forall$ — quantifier nesting

The tests were conducted on a 1.7GHz Intel Core i5 machine with 4GB of RAM, running OS X 10.7.5. The version of Z3 used was 4.2, compiled for 64-bit Intel architecture (using `gcc` 4.2, LLVM). The solving time reported is wall clock time of the execution of Z3.

	Formula size						Solving
Benchmark	P,Q		I		VC		time
	#	$\forall$	#	$\forall$	#	$\forall$	(Z3)
SLL: reverse	2	2	11	2	133	3	57ms
SLL: filter	5	1	14	1	280	4	39ms
SLL: create	1	0	1	0	36	3	13ms
SLL: delete	5	0	12	1	152	3	23ms
SLL: deleteAll	3	2	7	2	106	3	32ms
SLL: insert	8	1	6	1	178	3	17ms
SLL: find	7	1	7	1	64	3	15ms
SLL: last	3	0	5	0	74	3	15ms
SLL: merge	14	2	31	2	2255	3	226ms
SLL: rotate	6	1	-	-	73	3	22ms
SLL: swap	14	2	-	-	965	5	26ms
DLL: fix	5	2	11	2	121	3	32ms
DLL: splice	10	2	-	-	167	4	27ms

7.3 Buggy Examples

We also applied the tool to erroneous programs and programs with incorrect assertions. The results, including run-time statistics and formula sizes, are reported in Table 7. In addition, we measured the size of the model generated, by observing the size of the generated domain—which reflects the number of nodes in the heap. As expected, Z3 was able to produce concrete counterexample of a small size. Since these are slight variations of the correct programs, size and running time statistics are similar.

An example of generated output when a program fails to verify can be seen, for the $insert$ program, in Fig. 3. The tool reports, as part of its output, that counterexample occurs when $j =$ *null* and $h.val = i.val = e.val$.

```
Node insert(Node h, Node e) {
  Node i = h, j = null;
  while (i != null && e.val >= i.val) {
    j = i; i = i.n;
  }
  if (j != null) { j.n = e; e.n = i; }
  else { e.n = h; h = e; }
  return h;
}
```

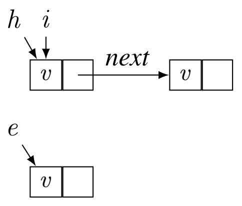

Fig. 3. Sample counterexample generated for a buggy version of $insert$. Here, the loop invariant required that $\forall\alpha : (h\langle next^*\rangle\alpha \land \neg i\langle next^*\rangle\alpha) \rightarrow \alpha <_{val} e$ (where $<_{val}$ is an ordering on nodes according to their values), but the loop will execute one more time, violating this.

Benchmark	Nature of Defect	Formula size						Solving time	C.e. size
		P,Q		I		VC		(Z3)	($\|L\|$)
		#	∀	#	∀	#	∀		
SLL: find	*null* pointer dereference.	7	1	7	1	64	3	18ms	2
SLL: deleteAll	Loop invariant in annotation is too weak to prove the desired property.	3	2	5	2	68	3	58ms	5
SLL: rotate	Transient cycle introduced during execution.	6	1	-	-	109	3	25ms	3
SLL: insert	Unhandled corner case when an element with the same value already exists in the list — ordering violated.	8	1	6	1	178	3	33ms	4

Table 7. Information about benchmarks that demonstrate detection of several kinds of bugs in pointer programs. In addition to the previous measurements, the last column lists the size of the generated counterexample in terms of the number of vertices, or linked-list nodes.

8 Discussion

8.1 Related Work

Decidable Logic. The results in this paper show that reachability properties of programs manipulating linked lists can be verified using a simple decidable logic AE^R. Many recent decidable logics for reasoning about linked lists have been proposed [17, 22, 15, 3]. In comparison to these works we drastically restrict the way quantifiers are allowed but permit arbitrary use of relations. Thus, strictly speaking our logic is incomparable to the above logics. We show that relations are used even in programs like *reverse* to write procedure summaries such as the one in (8) and for expressing numeric orders in sorting programs.

Employing Theorem Provers. The seminal paper on program verification [18] provides useful axioms for verifying reachability in linked data structures using theorem provers and conjectures that these axioms are complete for describing reachability. Lev-Ami et al. [14] show that no such axiomatization is possible. The current submission sidesteps the above impossibility results by restricting first order quantifications and by using the fact that Bernays-Schönfinkel formulas have finite model property.

Lahiri and Qadeer [13] provide rules for weakest of preconditions for programs with circular linked lists. The formulas are similar to Hesse's [10] but require that the programmer explicitly break the cycle. Our framework can be used both with and without the help of the programmer. In practice it may be beneficial to require that the programmer breaks the cycle in certain cases in order to allow invariants which distinguish between segments in the cycle.

Descriptive Complexity. Descriptive complexity was recently incorporated into the TVLA shape analysis framework [20]. In this paper we pioneer the use of descriptive complexity for *guaranteeing* that if the programmer writes AF^R assertions and if the program manipulates singly- and doubly-linked lists, then the VCs are guaranteed to be expressible as AE^R formulas.

8.2 Conclusion

The results in this paper shed some light on the complexity of reasoning about programs that manipulate linked data structures such as singly- and doubly-linked lists. The invariants in many of these programs can be expressed without quantifier alternation. Alternations are introduced by unbounded cutpoints and reasoning about more complicated directed acyclic graphs. Furthermore, for programs manipulating general graphs higher order reasoning may be required.

References

1. SMTLIB: Satisfiability modulo theories library. `http://smtlib.cs.uiowa.edu/docs.html`.
2. Technical report. `http://www.cs.tau.ac.il/~shachar/dl/tr-2013.pdf`.
3. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In ATVA, 2012.
4. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In TACAS, 2008.
5. C. Demetrescu and G. F. Italiano. Decremental all-pairs shortest paths. Encyclopedia of Algorithms, 2008.
6. G. Dong and J. Su. Incremental maintenance of recursive views using relational calculus/sql. SIGMOD Record, 29:44–51, 2000.
7. C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In POPL, 2001.
8. M. Frade and J. Pinto. Verification conditions for source-level imperative programs. Computer Science Review, 5(3):252–277, 2011.
9. J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In TACAS, 1995.
10. W. Hesse. Dynamic computational complexity. PhD thesis, Dept. of Computer Science, University of Massachusetts, Amherst, MA, 2003.
11. N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In CSL, 2004.
12. H. Kautz and B. Selman. Knowledge compilation and theory approximation. J. ACM, 43(2):193–224, 1996.
13. S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In POPL, 2008.
14. T. Lev-Ami, N. Immerman, T. W. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. Logical Methods in Computer Science, 5(2), 2009.
15. P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In POPL, 2011.
16. J. McCarthy. Towards a mathematical science of computation. In IFIP Congress, pages 21–28, 1962.
17. A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In PLDI, 2001.
18. G. Nelson. Verifying reachability invariants of linked structures. In POPL, 1983.
19. R. Piskac, L. M. de Moura, and N. Bjørner. Deciding effectively propositional logic using dpll and substitution sets. J. Autom. Reasoning, 44(4):401–424, 2010.
20. T. W. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. ACM Trans. Program. Lang. Syst., 32(6), 2010.
21. N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In POPL, 2005.
22. G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. J. Log. Algebr. Program., 73(1–2):111–142, 2007.