Verifying Properties of Well-Founded Linked Lists

Shuvendu K. Lahiri
shuvendu@microsoft.com
Microsoft Research

Shaz Qadeer
qadeer@microsoft.com
Microsoft Research

July 27, 2005

Technical Report
MSR-TR-2005-97

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Verifying Properties of Well-Founded Linked Lists

Shuvendu K. Lahiri
shuvendu@microsoft.com
Microsoft Research

Shaz Qadeer
qadeer@microsoft.com
Microsoft Research

Abstract

We describe a novel method for verifying programs that manipulate linked lists, based on two new predicates that characterize reachability of heap cells. These predicates allow reasoning about both acyclic and cyclic lists uniformly with equal ease. The crucial insight behind our approach is that a circular list invariably contains a *distinguished head cell* that provides a handle on the list. This observation suggests a programming methodology that requires the heap of the program at each step to be *well-founded*, i.e., for any field `f` in the program, every sequence $u.\mathtt{f}, u.\mathtt{f}.\mathtt{f}, \ldots$ contains at least one head cell. We believe that our methodology captures the most common idiom of programming with linked data structures. We enforce our methodology by automatically instrumenting the program with updates to two auxiliary variables representing these predicates and adding assertions in terms of these auxiliary variables.

To prove program properties and the instrumented assertions, we provide a first-order axiomatization of our two predicates. We also introduce a novel induction principle made possible by the well-foundedness of the heap. We use our induction principle to derive from two basic axioms a small set of additional first-order axioms that are useful for proving the correctness of several programs.

We have implemented our method in a tool and used it to verify the correctness of a variety of nontrivial programs manipulating both acyclic and cyclic singly-linked lists and doubly-linked lists.

1 Introduction

Program verification has made enormous progress in the last few decades. A significant contribution to this progress has been made by reasoning techniques based on first-order logic. For a number of programs, the verification condition whose validity establishes the correctness of the program is expressible in a combination of first order theories such as arithmetic, uninterpreted functions, and propositional logic. The pioneering work of Nelson and Oppen [27] described a method for combining decision procedures for individual theories to yield a decision procedure for the combined theory. Based on their results, a number of powerful automated theorem provers [9, 4] have been developed for program verification. Many successful tools [11, 3, 16] have used these provers for proving program properties.

However, first-order reasoning typically breaks down when we want to prove properties of programs that manipulate heap-allocated linked data structures. The main problem with reasoning about a data structure, such as a linked list, is that it is impossible to express an invariant about all members of a list in first-order logic. To achieve such a specification in general requires the use of the *reachability predicate* which cannot be expressed in first-order logic. Consequently, researchers have investigated richer logics such as combination of first order logic with transitive closure [17] and monadic second-order logic [25]. These approaches are typically unable to harness the advances made in automated theorem proving based on first-order logic.

In this paper, we develop a method for verifying linked data structures based entirely on first-order logic. The idea behind our approach is to provide a first-order approximation of the reachability predicate. Nelson [28] and Lev-Ami et al. [21] have also proposed first-order axiomatizations for linked lists. Our work improves upon these previous approaches in several significant ways. Most notably, our work makes the analysis of cyclic linked lists as uniform as acyclic lists. While theoretically incomplete, we believe that our approach is complete enough for most realistic programs. We have validated this belief by building a tool to mechanize our method. We have used our tool to verify a number of programs manipulating singly-linked and doubly-linked lists.

In the next section, we illustrate the benefits of our analysis for reasoning uniformly about acyclic and cyclic lists.

1.1 Motivation

The novelty of our technique is best illustrated by an example. Consider the following function `acyclic_simple` that iterates over an acyclic list pointed to by the variable `head` and sets the `data` field of each list element to 0. This example is representative of a variety of programs that use linked lists as sets and iterate over them.

```
//@ requires head != null
void acyclic_simple() {
     iter = head;
     while (iter != null) {
          iter.data = 0;
          iter = iter.next;
     }
}
```

To write the loop invariant for this program, we require the reachability function R, which maps each cell u to the set containing u and all cells reachable from u by following

the `next` field and excluding *null*. The cornerstone of all previous approaches to first-order reasoning about linked lists is the following axiom about R:

$$v \in R(u) \Leftrightarrow (v = u \vee (u.\texttt{next} \neq null \wedge v \in R(u.\texttt{next})))$$

This axiom and the rest of this section assumes, for simplicity but without loss of generality, that $null.\texttt{next} = null$. We have also added the precondition $\texttt{head} != null$ to make this example similar in as many ways as the example of a cyclic list traversal described later in the section.

Using R, the loop invariant for the program can be expressed as follows:

$$\forall v \in R(\texttt{head}) : (\texttt{iter} \neq null \wedge v \in R(\texttt{iter})) \vee v.\texttt{data} = 0$$

This loop invariant easily proves the following desired postcondition of the function.

$$\forall v \in R(\texttt{head}) : v.\texttt{data} = 0$$

The verification condition for this program is easily constructed using the weakest-precondition transformer [10], and the verification condition can be easily proved from the axiom using purely first-order reasoning based on uninterpreted functions and quantifier instantiation. We encourage the reader to perform this simple but interesting calculation.

Unfortunately, this simple approach breaks down with cyclic lists which are quite common, and perhaps the most common linked data structure used in operating system kernels. To illustrate the problem, consider the following variant `cyclic_simple` of the first function that iterates over a cyclic list pointed to by `head` and sets the `data` fields of the list elements to 0. The complication, in this case, is that the `next` field of the last element of the list points to `head` rather than *null*.

```
//@ requires head != null
void cyclic_simple() {
     head.data = 0;
     iter = head.next;
     while(iter !=head) {
          iter.data = 0;
          iter = iter.next;
     }
}
```

The circularity of the list results in a breakdown of the reasoning described earlier. The value of $R(\texttt{head})$ and $R(\texttt{iter})$ remains constant and equal to the set of all elements in the list during the entire execution of the loop. As a result, we are unable to write a simple loop invariant using R. Even proving the absence of null dereferences in `cylic_simple` is nontrivial compared to `acyclic_simple` where it is trivial because of the loop entry condition. In the remainder of this section, we show how our approach makes verification of programs manipulating cyclic lists as simple as those manipulating acyclic lists.

The crucial insight behind our approach is that a circular list invariably contains a *distinguished head cell*. This head cell provides a handle on the list, usually marks the beginning of the list, and the last element of the list points to it. In the program above, this cell is pointed to by the program variable `head`. Moreover, it is the presence of this cell that ensures the termination of iterations over circular linked lists. Let H denote the set of head cells of a program. Then, the above program has the precondition $\{\texttt{head}, null\} \subseteq H$. We think of *null* as a special head and consequently require that H always contain *null*. The usefulness of constraining H in this fashion will become clear later in this section.

We introduce a new axiomatization of linked lists with two new functions, R_H and Id_H. R_H maps each cell u to the set of cells containing u and all cells reachable from u by following the `next` field until a cell $v \in H$ is reached. The cell v is not included in $R_H(u)$. Finally, we define $Id_H(u)$ to be v. The cell v acts as a block to the traversal of the `next` field from u and its identity is captured as $Id_H(u)$. We call $Id_H(u)$ the *blocking cell* for u. The axioms for R_H and Id_H are as follows:

$$v \in R_H(u) \Leftrightarrow (u = v \vee (u.\texttt{next} \notin H \wedge v \in R_H(u.\texttt{next})))$$
$$Id_H(u) = (u.\texttt{next} \in H)\ ?\ u.\texttt{next} : Id_H(u.\texttt{next})$$

The first axiom is essentially the definition of R_H and is similar in spirit to the axiom for R described earlier. The second axiom says that $Id_H(u)$ is $u.\texttt{next}$ if $u.\texttt{next} \in H$ and $Id_H(u.\texttt{next})$ otherwise.

Using the definition of Id_H, we can specify that `head` points to a circular list by adding the following precondition to `cyclic_simple`.

```
//@ requires Id_H(head) = head
```

From the definition of R_H and Id_H, it is clear that if $Id_H(u) = u$, then $u \in H$ and $R_H(u)$ is the set of cells in a cycle in which no cell other than u is in H. Therefore, the set of cells in the circular list pointed to by `head` is given by $R_H(\texttt{head})$. Using the definition of R_H and Id_H, we can now write the loop invariant for the loop in `cyclic_simple` as follows.

$$\begin{array}{l}\forall v \in R_H(\texttt{head}) : \\ \qquad (\texttt{iter} \neq \texttt{head} \wedge v \in R_H(\texttt{iter})) \vee v.\texttt{data} = 0 \\ \wedge\ Id_H(\texttt{iter}) = \texttt{head}\end{array}$$

Observe that the first conjunct of the loop invariant is similar in structure to the loop invariant for the function `acyclic_simple`. The two new axioms allow us to prove the correctness of the loop invariant, the absence of null dereferences, and the following desired postcondition for `cyclic_simple`:

$$\forall v \in R_H(\texttt{head}) : v.\texttt{data} = 0$$

As mentioned earlier, proving the absence of null dereferences in `cyclic_simple` is nontrivial and requires the use of the axiom about Id_H.

Our new axiomatization is a generalization of the first axiomatization that worked only for acyclic lists. If $H = \{null\}$, the axiom about R_H reduces to the axiom about R. Indeed, we can verify the first program using the new axiomatization by introducing a precondition $H = \{null\}$ and rewriting the loop invariant to

$$\begin{array}{l}\forall v \in R_H(\texttt{head}) : \\ \qquad (\texttt{iter} \neq null \wedge v \in R_H(\texttt{iter})) \vee v.\texttt{data} = 0\end{array}$$

Thus, our new axiom system can verify programs manipulating acyclic as well as cyclic lists with equal ease.

1.2 Contributions

The main technical contribution of this paper is a novel method for verifying linked lists based on two new predicates

that characterize reachability of heap cells. These predicates allow reasoning about both acyclic and cyclic lists uniformly with equal ease. The crucial insight behind our approach is that a circular list invariably contains a *distinguished head cell* that provides a handle on the list. This observation suggests a programming methodology that requires the heap of the program at each step to be *well-founded*, i.e., for any field f in the program, every sequence $u.\mathtt{f}, u.\mathtt{f}.\mathtt{f}, \ldots$ contains at least one head cell. The set of head cells is identified by a new variable added to the program by the programmer.

We believe that our methodology captures the most common idiom of programming with linked data structures. We enforce our methodology by automatically instrumenting the program with updates to two auxiliary variables representing these predicates and adding assertions in terms of these auxiliary variables. Our instrumentation captures well-foundedness *precisely* — the instrumented program fails one of these assertions if and only if the original program reaches a state containing a heap that is not well-founded.

To prove program properties and the instrumented assertions, we provide a first-order axiomatization of our two predicates. Our axiomatization consists of the two basic first-order axioms introduced in Section 1.1. We also introduce a novel induction principle made possible by the well-foundedness of the heap. We use our induction principle to derive from the two basic axioms a small set of additional first-order axioms that are useful for proving the correctness of several programs. All of these derived axioms are intuitive and natural and we state them precisely in Section 5.

We have implemented our method in a tool and used it to verify the correctness of a variety of nontrivial programs manipulating both acyclic and cyclic singly-linked lists and doubly-linked lists. Our approach allows us to leverage not only powerful first-order theorem provers but also invariant inference engines based on indexed predicate abstraction [12, 19]. We describe our preliminary experience with this tool in Section 6.3.

2 Examples

In Section 1.1, we illustrated our method by verifying the functions `acyclic_simple` and `cyclic_simple`. These functions were interesting and illustrative but comparatively simple. In this section, we illustrate our method on three more complex examples. The first example in Section 2.1 modifies the link structure of the heap; the second example in Section 2.2 uses arithmetic in addition to heap manipulation; the third example in Section 2.3 manipulates a doubly-linked list. We note that the examples in Section 1.1 and all examples of this section have been verified by the tool whose details we provide in Section 6.

2.1 Set-union

Our first example is the function `union` shown in Figure 1. This example is taken from a paper by Nelson [28]. The function `union` takes two circular linked lists `a` and `b` as arguments. Each list represents a set; the field `r` in a cell contains the identifier of the unique set to which it belongs.

The function `union` has a number of preconditions also stated in Figure 1. The second precondition says that both `a` and `b` are pointing to head cells. The third precondition says that `a` and `b` are pointing to circular lists. The fourth precondition says that the lists pointed to by `a` and `b` are disjoint.

```
//@ requires a != null && b != null
//@ requires {null, a, b} ⊆ H_next
//@ requires Id_next(a) == a && Id_next(b) == b
//@ requires R_next(a) ∩ R_next(b) == {}
//@ requires ∀u ∈ R_next(b) : u.r == b.r

//@ ensures Id_next(a) = b && Id_next(b) = a
//@ ensures ∀u ∈ (R_next(a) ∪ R_next(b)) : u.r == b.r
//@ ensures
     (R_next(a) ∪ R_next(b)) == (old(R_next)(a) ∪ old(R_next)(b))
//@ ensures ∀u ∉ old(R_next)(a) : u.r = old(u.r)

void union(Cell a, Cell b) {
     a.r = b.r;
     Cell curr = a.next;

     while (curr != a) {
          curr.r = b.r;
          curr = curr.next;
     }

     Cell tmp = a.next;
     a.next = b.next;
     b.next = tmp;
}
```

Figure 1: Performing the union of two circular lists

The objective of `union` is to merge the list pointed to by `a` into the list pointed to by `b`. The function `union` first sets the `r` field of each cell in `a` to `b.r`. Finally, the contents of `a.next` and `b.next` are swapped to merge the two lists.

Our tool automatically instruments `union` with updates to the functions $\mathtt{Id}_{\mathtt{next}}$ and $\mathtt{R}_{\mathtt{next}}$. This instrumentation is introduced only for the last two statements, which are the only statements that update the `next` field. For each of these statements, the instrumentation also checks by inserting an assertion that the update of `next` leaves the heap well-founded. For example, the assertion introduced just before the last statement of the function is as follows:

$$\mathtt{assert}(\mathtt{b} \in \mathtt{R}_{\mathtt{next}}(\mathtt{tmp}) \Rightarrow \mathtt{tmp} \in \mathtt{H}_{\mathtt{next}})$$

This assertion fails precisely if there is a chain of cells connected by `next` from `tmp` to `b` in which no cell is in $\mathtt{H}_{\mathtt{next}}$. Such a chain causes an ill-founded cycle of cells upon execution of the statement `b.next = tmp`. Furthermore, this statement modifies $\mathtt{Id}_{\mathtt{next}}(\mathtt{b})$ and $\mathtt{R}_{\mathtt{next}}(\mathtt{b})$ as follows:

$$\begin{array}{lcl} \mathtt{Id}_{\mathtt{next}}(\mathtt{b}) & = & (\mathtt{tmp} \in \mathtt{H}_{\mathtt{next}}) \ ? \ \mathtt{tmp} \ : \ \mathtt{Id}_{\mathtt{next}}(\mathtt{tmp}) \\ \mathtt{R}_{\mathtt{next}}(\mathtt{b}) & = & (\mathtt{tmp} \in \mathtt{H}_{\mathtt{next}}) \ ? \ \{\mathtt{b}\} \ : \ \{\mathtt{b}\} \cup \mathtt{R}_{\mathtt{next}}(\mathtt{tmp}) \end{array}$$

A precise and complete description of both $\mathtt{Id}_{\mathtt{next}}$ and $\mathtt{R}_{\mathtt{next}}$ is given in Section 4.

For `union`, we prove the absence of null dereferences, the assertions described above to check that the heap remains well-founded, and three postconditions. The first postcondition is particularly interesting. It states that the blocking cell for `a` is `b` and the blocking cell for `b` is `a`. These two facts together mean that cells `a` and `b` are the only two head cells in a cycle, which indicates indirectly that indeed the union of the two lists has been created. In the third postcondition, $\mathtt{old}(\mathtt{R}_{\mathtt{next}})$ refers to the value of $\mathtt{R}_{\mathtt{next}}$ at the beginning of the function. The final postcondition says that the `r` field

```
//@ requires {null} ⊆ H_next
//@ requires Id_next(l) == null
//@ requires l != null && p != null
//@ requires p ∉ H_next
//@ requires p.data > l.data
//@ requires sorted(l)
//@ requires p ∉ R_next(l)

//@ ensures sorted(l)
//@ ensures p ∈ R_next(l)

void insert(Cell l, Cell p) {
     Cell curr = l;
     Cell succ = l.next;

     while (succ != null) {
          if (p.data > succ.data) {
               curr = succ;
               succ = curr.next;
          }
          else
               break;
     }

     p.next = succ;
     curr.next = p;
}
```

Figure 2: Inserting an element into a sorted acyclic list

remains unchanged for all those cells which do not belong to $\mathtt{R_{next}(a)}$ initially.

So far, the set of head cells has been a constant. The first postcondition also motivates the need to modify the set of head cells. It would be intuitively more satisfactory if the programmer can remove the cell `a` from $\mathtt{H_{next}}$ once the list `a` has been merged into `b`. Therefore, we allow the programmer to remove a cell from $\mathtt{H_{next}}$ by using the `Remove` operation. For example, we could add the statement

$$\mathtt{H_{next}.Remove(a);}$$

at the very end of `union`. With this modification, our tool proves the following more pleasing postconditions.

```
//@ ensures Id_next(b) = b && Id_next(a) = b
//@ ensures ∀u ∈ R_next(b) : u.r == b.r
//@ ensures R_next(b) == (old(R_next)(a) ∪ old(R_next)(b))
```

Since the values of the variables $\mathtt{Id_{next}}$ and $\mathtt{R_{next}}$ depend on $\mathtt{H_{next}}$, the instrumentation for this statement is nontrivial. Since $\mathtt{H_{next}}$ becomes smaller, the heap might not remain well-founded. The instrumentation not only checks via an assertion that the heap remains well-founded, but also updates the values of $\mathtt{Id_{next}}$ and $\mathtt{R_{next}}$ appropriately. We give a precise description of the instrumentation for this statement as well as for $\mathtt{H_f.Add(x)}$, the converse of this statement, in Section 4.

2.2 Insert

Our second example is the function `insert` shown in Figure 2. The function `insert` takes an acyclic list `l` and a cell `p` as arguments. The cells in the list `l` are in sorted order based on the values of the field `data` of the cells. The predicate `sorted(l)` is defined as follows:

$$\mathtt{sorted(l)} = \forall u \in \mathtt{R_{next}(l)} : u.\mathtt{next} == null \,||\, u.\mathtt{data} <= u.\mathtt{next.data}$$

The objective of `insert` is to insert the cell `p` in the appropriate place in `l` so that `l` remains sorted. The fourth precondition requires that the cell `p` is not in $\mathtt{H_{next}}$. If $\mathtt{p} \in \mathtt{H_{next}}$, then on return, $\mathtt{R_{next}(l)}$ contains all the cells from `l` upto but not including `p`, and thus violates the the second postcondition $\mathtt{p} \in \mathtt{R_{next}(l)}$. The fifth precondition of `insert` is also worth explaining. To simplify the coding of `insert`, it is expected that the first element of `l` is a dummy whose `data` field is guaranteed to be less than any value that might be inserted in the list.

This example illustrates an important advantage of axiomatizing linked lists in first-order logic. The specification of `insert` uses a combination of facts about reachability via the `next` field, uninterpreted functions, arithmetic, and propositional logic. The axiomatization of linked lists in first-order logic immediately allows us to use any one of a number of theorem provers that deal with a combination of first-order theories. For our implementation, we are using the UCLID theorem prover [7]. Although this example does not use any arrays, adding them is a simple matter because they can be modeled easily with uninterpreted functions and the well-known called select-update axioms.

```
//@ requires wf_dlist_head(hd)
//@ requires p != null
//@ requires p ∈ R_next(hd)
//@ requires p != hd
//@ requires Id_next(p) == hd

//@ ensures p ∉ R_next(hd)
//@ ensures wf_dlist_head(hd)

void dlist_remove(Cell hd, Cell p) {
     Cell tp = p.prev;
     Cell tn = p.next;
     tp.next = tn;
     tn.prev = tp;
}
```

Figure 3: Removing an element from a doubly-linked list

2.3 Remove

Our third example is the function `dlist_remove` given in Figure 3. This function removes a cell `p` from a cyclic doubly-linked list with head `hd`. This example illustrates that our technique handles doubly-linked lists just as cleanly as singly-linked lists.

Our instrumentation adds variables $\mathtt{R_{next}}$ and $\mathtt{Id_{next}}$ for the linking field `next`, and $\mathtt{R_{prev}}$ and $\mathtt{Id_{prev}}$ for the linking field `prev`. The instrumentation happens just as before for each linking field as if they are independent. However since the linking field pair (`next`, `prev`) forms a doubly linked list, we need to define additional preconditions to relate the two fields and their auxiliary variables.

Domains

		Boolean	=	$\{false, true\}$
		Integer	=	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
u, v, w	$\in$	*Cell*	=	$\{null, \ldots\}$
`x, y, z`	$\in$	*Var*		
`f, g, h`	$\in$	*Field*		
c	$\in$	*Const*	=	$Boolean \cup Integer \cup \{null\}$
		Op	=	{ `+`, `-`, `==`, `!=`, `<`, `&&`, `||` }
s	$\in$	*Stmt*	::=	`x = c` | `x = y` | `x = y` *Op* `z` | `assume(x)` | `assert(x)` | `x = new` | `x = y.f` | `x.f = y`

Figure 4: Program Syntax.

We define a predicate `wf_dlist_head(hd)` to denote that `hd` points to a well formed doubly linked list. This predicate is a conjunction of the following predicates:

1. `hd != null`
2. `hd` $\in$ $\mathtt{H}_{\mathtt{next}}$ `&&` `hd` $\in$ $\mathtt{H}_{\mathtt{prev}}$
3. $\forall u \in \mathtt{R}_{\mathtt{next}}(\mathtt{hd}) : u.\mathtt{next.prev} == u$ `&&` $u.\mathtt{prev.next} == u$
4. $\mathtt{R}_{\mathtt{next}}(\mathtt{hd}) == \mathtt{R}_{\mathtt{prev}}(\mathtt{hd})$
5. $\mathtt{Id}_{\mathtt{next}}(\mathtt{hd}) == \mathtt{hd}$ `&&` $\mathtt{Id}_{\mathtt{prev}}(\mathtt{hd}) == \mathtt{hd}$

The last conjunct indicates that `hd` is the unique head cell in $\mathtt{H}_{\mathtt{next}}$ and $\mathtt{H}_{\mathtt{prev}}$ present in the cyclic list. The predicate `wf_dlist_head(hd)` is particularly useful when the routine `dlist_remove` is invoked in a loop to, for example, remove all the cells from the list that satisfy some property. Then this predicate will be a conjunct in the loop invariant.

The first precondition requires that the list is well formed. The next two preconditions state that `p` points to a cell in the list. The next precondition states that `p` is different from `hd` and is required otherwise the list becomes ill-founded after the remove operation. The postconditions assert that `p` is removed from the list and the list remains well-formed.

3 Programs

The set *Var* is a set of program variables. A variable in *Var* may have one of three types: *Boolean*, *Integer*, or *Cell*. The set *Cell* contains the addresses of heap objects, each of which may have fields from the set *Field*. A field may also have one of the three aforementioned types. A field of type *Boolean* is a map from *Cell* to *Boolean*, a field of type *Integer* is a map from *Cell* to *Integer*, and a field of type *Cell* is a map from *Cell* to *Cell*. We refer to the last category of fields as reference fields. Finally, *Const* is the set of constants that may appear in the program. The constant *null* is a special constant of type *Cell*.

A program is a *control flow graph* (PC, E, pc_i, pc_f, L) with five components. PC is a finite set of program locations, $E \subseteq PC \times PC$ is the set of control-flow edges, $pc_i \in PC$ is the initial location where program execution begins, $pc_f \in PC$ is the final location where program execution terminates, and L is a function that maps each edge in E to a statement in *Stmt*.

The restricted syntax of the statements in *Stmt* does not result in any loss of generality, because more complex statements can reduced to simple statements by the introduction of temporary variables. The statement `assume(x)` together with nondeterminism in the control flow graph can be used to encode *if-then-else* and *while* statements. Moreover, other operations can be encoded with the operations in *Op*. For example, boolean negation of `x` can be encoded as `x ==` *false*. We also assume that the program is free of typing errors.

3.1 Programming with heads

We allow programmers to specify a subset of the reference fields as *linking* fields. These fields are expected to provide the links in data structures such as singly-linked and doubly-linked lists. For each linking field `f`, the programmer is allowed access to a variable $\mathtt{H}_{\mathtt{f}}$ of type $Set(Cell)$. The set $\mathtt{H}_{\mathtt{f}}$ contains a subset of the set of allocated cells that together make all lists linked by `f` well-founded. The set $\mathtt{H}_{\mathtt{f}}$ is required to always contain the cell *null*. We provide two operations to update $\mathtt{H}_{\mathtt{f}}$.

1. $\mathtt{H}_{\mathtt{f}}$`.Add(x)`: The variable `x` is required to be of type *Cell*. This statement adds the cell obtained by evaluating `x` to the set $\mathtt{H}_{\mathtt{f}}$. This statement provides the fundamental mechanism for making circular lists linked by the field `f` well-founded. A programmer creates such a list by first creating the head cell and then adding it to the set $\mathtt{H}_{\mathtt{f}}$.
2. $\mathtt{H}_{\mathtt{f}}$`.Remove(e)`: Again, the variable `x` is required to be of type *Cell*. This operation requires that `x` be different from *null*. This statement removes the cell obtained by evaluating `x` from the set $\mathtt{H}_{\mathtt{f}}$. The precondition ensures that we never remove *null* from $\mathtt{H}_{\mathtt{f}}$. We have already illustrated the utility of this operation in Section 2.1. This operation is used to remove the head cells of those circular linked lists that are merged into other lists.

In addition to the usual statements, we also allow the edges of the control flow graph to be labelled with the above operations on $\mathtt{H}_{\mathtt{f}}$ for each linking field `f`.

3.2 Semantics

The state σ of the program contains a program counter in PC and a valuation of the variables in *Var*, the fields in *Field*, and the head variables $\mathtt{H}_{\mathtt{f}}$ for each linking field `f`. Variables of type *Boolean*are initialized to *false*, variables of type *Integer*are initialized to 0, and variables of type *Cell*are initialized to *null*.

The state also contains a special variable `Alloc` of type $Set(Cell)$ to model memory allocation via `new`. The variable `Alloc` is initialized to {}. The statement `x = new` removes a nondeterministically chosen cell that is not equal to *null* from $Cell \setminus$ `Alloc`, assigns it to `x`, and adds it to `Alloc`. Thus, the statement `x = new` is desugared to the following statements:

```
x  =  choose(Cell \ {null}\Alloc);
Alloc  =  Alloc ∪ {x};
```

When the program executes, its state changes in accordance with the standard operational semantics of its control flow graph. To model the program misbehaving by performing a dereference of *null* or by failing an assertion, we add a special state *wrong* with no transition out of it. For example, if the value of the program counter in σ is l and the edge

$(l, l') \in E$ is labelled with $\mathtt{x.f} = \mathtt{y}$, then one of the following may happen.

1. $\sigma(\mathtt{x})$ is *null* and the program moves to the state *wrong*.
2. $\sigma(\mathtt{x})$ is not *null* and the program makes a transition to a state σ' in which the program counter is l', the map for $\mathtt{f}$ is modified only at cell $\sigma(\mathtt{x})$ to $\sigma(\mathtt{y})$, and otherwise the state remains unchanged.

The operational semantics of the other statements can be defined similarly.

4 Program instrumentation

In this section, we show how to automatically instrument a program to ensure that the linked lists in the heap always remain well-founded. The instrumentation is performed with respect to a subset of reference fields called *linking* fields that act as the links in a linked structure. We automatically instrument the program with two auxiliary variables for every linking field $\mathtt{f}$. These variables, called $\mathtt{R_f}$ and $\mathtt{Id_f}$, record information about the shape of the heap graph and are essential to our verification method. The variable $\mathtt{R_f}$ is a map from *Cell* to *Set*(*Cell*). The variable $\mathtt{Id_f}$ is a map from *Cell* to *Cell*. Intuitively, for any $u \in Cell$, $\mathtt{R_f}(u)$ is the set of heap cells containing u and every cell obtained by one or more applications of $\mathtt{f}$ to u until a cell $v \in \mathtt{H_f}$ is reached and $\mathtt{Id_f}(u) = v$. Note that the final cell v is not a member of $\mathtt{R_f}(u)$. We will often write $\mathtt{R_f}(u, v)$ to denote $v \in \mathtt{R_f}(u)$.

Although the variables $\mathtt{R_f}$ and $\mathtt{Id_f}$ can be defined as a mathematical function of the program state, this definition uses transitive closure and therefore cannot be expressed in first-order logic. An important and surprising insight of our work is that even though these variables cannot be defined without using transitive closure, the *updates* to them as the program state changes can be defined entirely in first-order logic. We automatically instrument the program to record these update and are thus able to state the assertions which ensure that the heap remains well-founded.

Let P be a program with control flow graph (PC, E, pc_i, pc_f, L). The instrumented program $P^{\#}$ is obtained by instrumenting each individual statement $L(e)$ for each edge $e \in E$. We now define the instrumentation for a given statement s. In many cases shown below, the instrumented statement contains control flow. Such a complex statement is used only for clarity of presentation, and can easily be translated into a simple statement without control flow.

1. If the statement s is of the form `x = c`, `x = y`, `x = y` *Op* `z`, `assume(x)`, `assert(x)`, `x = new`, or `x = y.f`, then the instrumented statement is identical to s. The definition of $\mathtt{R_f}$ and $\mathtt{Id_f}$ depends only on the value of map $\mathtt{f}$ and the set $\mathtt{H_f}$. Since neither of those are changed by the statement, we do not need to update the instrumentation variables.

2. If the statement is of the form `x.f = y` and `f` is a linking field, then the instrumented statement is

```
assert(x != null);
assert(x ∈ R_f(y) => y ∈ H_f);
if (y ∈ H_f) {
    Id_f = λi. { y          if x ∈ R_f(i)
               { Id_f(i)    otherwise

    R_f = λi. { (R_f(i) \ R_f(x)) ∪ {x}   if x ∈ R_f(i)
              { R_f(i)                    otherwise
}
else {
    Id_f = λi. { Id_f(y)    if x ∈ R_f(i)
               { Id_f(i)    otherwise

    R_f = λi. { (R_f(i) \ R_f(x)) ∪ {x} ∪ R_f(y)   if x ∈ R_f(i)
              { R_f(i)                             otherwise
}
x.f = y;
```

   This statement is the only one that updates the link structure of the heap. First, the instrumentation checks via the assertion `assert(x ∈ R_f(y) => y ∈ H_f)` that the heap remains well-founded, that is, no cycle of cells unbroken by a member of $\mathtt{H_f}$ is created as a result of updating the $\mathtt{f}$ field of $\mathtt{x}$ to $\mathtt{y}$. If such a cycle is created, then there must be a path from from $\mathtt{y}$ to $\mathtt{x}$ in which no cell, including $\mathtt{y}$ and $\mathtt{x}$, is in $\mathtt{H_f}$, which results in a violation of this assertion.

   Second, the two instrumentation variables $\mathtt{Id_f}$ and $\mathtt{R_f}$ are updated. The values of these functions is updated at a particular cell i only if $\mathtt{x} \in \mathtt{R_f}(i)$, that is, there is a path from i to $\mathtt{x}$ not broken by any member of $\mathtt{H_f}$. If $\mathtt{x} \in \mathtt{R_f}(i)$, the update is split into two cases. If $\mathtt{y} \in \mathtt{H_f}$, then $\mathtt{Id_f}(i)$ becomes $\mathtt{y}$ and we remove from $\mathtt{R_f}(i)$ everything that is reachable from $\mathtt{x}$ without hitting a member of $\mathtt{H_f}$ but then add $\mathtt{x}$ itself. If $\mathtt{y} \notin \mathtt{H_f}$, then $\mathtt{Id_f}(i)$ becomes $\mathtt{Id_f}(\mathtt{y})$ and we remove from $\mathtt{R_f}(i)$ everything that is reachable from $\mathtt{x}$ but then add $\mathtt{x}$ itself and everything that is reachable from $\mathtt{y}$.

3. If the statement is of the form `H_f.Add(x)`, then the instrumented statement is

```
if (x ∉ H_f) {
    Id_f = λi. { x          if x ∈ R_f(i) && x != i
               { Id_f(i)    otherwise

    R_f = λi. { R_f(i) \ R_f(x)   if x ∈ R_f(i) && x != i
              { R_f(i)            otherwise

    H_f = H_f ∪ {x};
}
```

   This statement does not update the linking structure of the heap; it only changes the value of $\mathtt{H_f}$. Since the values of the instrumentation variables $\mathtt{Id_f}$ and $\mathtt{R_f}$ depend on $\mathtt{H_f}$, these variables must be updated if $\mathtt{H_f}$ changes in case $\mathtt{x} \notin \mathtt{H_f}$. The values of $\mathtt{Id_f}$ and $\mathtt{R_f}$ are updated at a particular cell i only if $\mathtt{x} \in \mathtt{R_f}(i) \wedge \mathtt{x} \neq i$, that is, $\mathtt{x}$ becomes new blocking cell for i. In this case, $\mathtt{Id_f}(i)$ becomes $\mathtt{x}$ and we remove from $\mathtt{R_f}(i)$ everything that is reachable from $\mathtt{x}$ without hitting a member of $\mathtt{H_f}$.

4. If the statement is of the form `H_f.Remove(x)`, then the instrumented statement is

```
if (x ∈ Hf) {
    assert(Idf(x) != x);

    Idf = λi. { Idf(x)   if Idf(i)== x
              { Idf(i)   otherwise

    Rf = λi. { Rf(i) ∪ Rf(x)   if Idf(i)== x
             { Rf(i)           otherwise

    Hf = Hf \ {x};
}
```

This statement, just like $\texttt{Hf.Add(x)}$, does not update the linking structure of the heap; it only changes the value of $\texttt{Hf}$. The values of the instrumentation variables $\texttt{Id}_\texttt{f}$ and $\texttt{R}_\texttt{f}$ must be updated if $\texttt{Hf}$ changes in case $\texttt{x} \in \texttt{Hf}$. First, the instrumentation checks via an assertion that the removal of x from the set of head cells does not result in a heap that is not well-founded. A bad heap can result if x is the only head cell in a cycle, a condition captured by $\texttt{Id}_\texttt{f}(\texttt{x}) = \texttt{x}$ and the assertion checks for precisely the negation of this condition. Note that since $\texttt{Id}_\texttt{f}(\mathit{null}) = \mathit{null}$ by definition, this assertion also checks the precondition of this operation that x is nonnull.

Second, the two instrumentation variables $\texttt{Id}_\texttt{f}$ and $\texttt{R}_\texttt{f}$ are updated. The values of $\texttt{Id}_\texttt{f}$ and $\texttt{R}_\texttt{f}$ are updated at a particular cell i only if $\texttt{Id}_\texttt{f}(i) = \texttt{x}$, that is, x is the blocking cell for i. In this case, $\texttt{Id}_\texttt{f}(i)$ becomes $\texttt{Id}_\texttt{f}(\texttt{x})$ and we add to $\texttt{R}_\texttt{f}(i)$ everything that is reachable from x without hitting a member of $\texttt{Hf}$.

4.1 Correctness

In this section, we formalize the correctness of our instrumentation. The instrumented state θ of the program is a valuation of the variables in $\texttt{R}_\texttt{f}$ and $\texttt{Id}_\texttt{f}$ for each linking field f. While the original program makes transitions of the form $\sigma \to \sigma'$, the instrumented program makes transitions of the form $(\sigma, \theta) \to (\sigma', \theta')$. To state our correctness theorem, we first need to define a well-formed state.

Definition 1 (Well-founded function) *A function f : Cell → Cell is well-founded with respect to a set of cells H, if for any cell $u \in$ Cell, there exists $n > 0$, such that $f^n(u) \in H$.*

If f is well-founded with respect to h, we can define the functions Id^f_H : Cell → Cell and R^f_H : Cell → Set(Cell) as follows.

$$R^f_H(u) = \{v \in \mathit{Cell} \mid v = u \vee \exists n : 0 < n : (v = f^n(u) \wedge \forall m : 0 < m \leq n : f^m(u) \notin H)\}$$

$$Id^f_H(u) = f^n(u), \text{where } n = \mathit{min}\{m \mid 0 < m \wedge f^m(u) \in H\}$$

Definition 2 (Well-founded state) *A state σ is* well-founded *if for every linking field f, the function $\sigma(\texttt{f})$ is well-founded with respect to $\sigma(\texttt{Hf})$.*

Definition 3 (Well-formed state) *A state σ is well-formed if the following conditions are satisfied:*

1. *σ is well-founded.*
2. *null $\notin$ Alloc.*
3. *For every variable x of type Cell, $\sigma(\texttt{x}) \in \sigma(\texttt{Alloc})$ or $\sigma(\texttt{x}) = \mathit{null}$.*
4. *For every cell $u \in \sigma(\texttt{Alloc})$ and reference field f $\in$ Field, $\sigma(\texttt{f})(u) \in \sigma(\texttt{Alloc})$ or $\sigma(\texttt{f})(u) = \mathit{null}$.*
5. *For every cell $u \in \mathit{Cell} \setminus \sigma(\texttt{Alloc})$ and reference field f $\in$ Field, $\sigma(\texttt{f})(u) = \mathit{null}$.*
6. *For every linking field f, null $\in \sigma(\texttt{Hf})$.*

A state is *ill-formed* if it is not well-formed.

We define an instrumentation function I on well-formed states. For every program state σ, the application $I(\sigma)$ yields an instrumented state θ such that $\theta(\texttt{Id}_\texttt{f}) = Id^f_H$ and $\theta(\texttt{R}_\texttt{f}) = R^f_H$, where $f = \sigma(\texttt{f})$ and $H = \sigma(\texttt{Hf})$. Now, we can state a theorem that characterizes the precision of our instrumentation.

Theorem 1 *Suppose σ and σ' are well-formed states of a program P and $P^\#$ is the instrumented version of P. Then the following statements are true.*

1. *P can make a transition from σ to σ' iff $P^\#$ can make a transition from $(\sigma, I(\sigma))$ to $(\sigma', I(\sigma'))$.*
2. *P makes a transition from σ to either an ill-formed state or wrong iff $P^\#$ makes a transition from $(\sigma, I(\sigma))$ to wrong.*

Proof: We can prove the theorem by a case analysis over the various types of statements.

1. Suppose the statement s is of the form x = c, x = y, x = y *Op* z, assume(x), assert(x), or x = y.f. Then the instrumented statement is identical to s. Since s neither allocates any cell nor modifies a linking field of any cell, the proof follows easily.

2. Suppose the statement s is of the form x = new. Then the instrumented statement is identical to s and as explained in Section 3 is desugared to

   ```
   x = choose(Cell \ {null}\Alloc);
   Alloc = Alloc ∪ {x};
   ```

   Let v be the value of x in σ'. Then $v \in \sigma'(\texttt{Alloc})$. For every reference field f, we have $\sigma'(\texttt{f})(v) = \sigma(\texttt{f})(v) = \mathit{null}$. Finally, since this operation does not modify a linking field of any cell, the state σ' remains well-founded.

3. Suppose the statement s is of the form x.f = y and f is a linking field. Then the instrumented statement is as described earlier in Section 4. If s goes wrong because of a null dereference on x, then the instrumented statement goes wrong because of the first assertion. Let $\texttt{x} = u$ and $\texttt{y} = v$ in the state σ. Let $H = \sigma(\texttt{Hf}) = \sigma'(\texttt{Hf})$. If a cycle that is not broken by any cell in H is created in σ', then there must be a path from v to u in σ in which no cell, including v and u, is in H. In this case, the second assertion in the instrumented statement goes wrong. Otherwise, the state σ' is well-formed. Let $f = \sigma(\texttt{f})$ and $f' = \sigma'(\texttt{f})$. Let θ and θ' be the instrumented state before and after the operation respectively. We know that $\theta(\texttt{Id}_\texttt{f}) = Id^f_H$ and $\theta(\texttt{R}_\texttt{f}) = R^f_H$. Let us fix $i \in \mathit{Cell}$. Then, we must

show that $\theta'(\mathtt{Id_f})(i) = Id_H^{f'}(i)$ and $\theta'(\mathtt{R_f})(i) = R_H^{f'}(i)$. We can prove this by a four-way case analysis over the values of the pair $(H(v), u \in \theta(\mathtt{R_f})(i))$ of Boolean predicates.

4. If the statement is of the form $\mathtt{H_f.Add(x)}$, then the instrumented statement is as described earlier in Section 4. Let $H = \sigma(\mathtt{H_f})$ and $H' = \sigma'(\mathtt{H_f})$. Let $f = \sigma(\mathtt{f}) = \sigma'(\mathtt{f})$. Let $\mathtt{x} = u$ in the state σ. Let θ and θ' be the instrumented state before and after the operation respectively. If $u \in H$, then $\sigma' = \sigma$ and $\theta' = \theta$ and we are done. Otherwise $u \notin H$. We know that $\theta(\mathtt{Id_f}) = Id_H^f$ and $\theta(\mathtt{R_f}) = R_H^f$. Let us fix $i \in \mathit{Cell}$. Then, we must show that $\theta'(\mathtt{Id_f})(i) = Id_{H'}^f(i)$ and $\theta'(\mathtt{R_f})(i) = R_{H'}^f(i)$. We can prove this by a two-way case analysis over the Boolean predicate $u \in \theta(\mathtt{R_f})(i) \wedge u \neq i$.

5. If the statement is of the form $\mathtt{H_f.Remove(x)}$, then the instrumented statement is as described earlier in Section 4. Let $H = \sigma(\mathtt{H_f})$ and $H' = \sigma'(\mathtt{H_f})$. Let $f = \sigma(\mathtt{f}) = \sigma'(\mathtt{f})$. Let $\mathtt{x} = u$ in the state σ. Let θ and θ' be the instrumented state before and after the operation respectively. If $u \notin H$, then $\sigma' = \sigma$ and $\theta' = \theta$ and we are done. Otherwise $u \in H$. If $u = \mathit{null}$, then since σ is well-formed, we have $u \notin \mathtt{Alloc}$ and $\sigma(\mathtt{f})(u) = \mathit{null}$. Therefore $\theta(\mathtt{Id_f})(u) = \mathit{null} = u$ and the first assertion in the instrumented statement goes wrong. Otherwise $u \in H \wedge u \neq \mathit{null}$. As a result of removing u from H, if a cycle that is not broken by any cell in H' is created in σ', then there must be a path from u to u in σ in which the only cell in H is u itself. Again, we get $\theta(\mathtt{Id_f})(u) = u$ and the first assertion in the instrumented statement goes wrong. Otherwise, we have that σ' is a well-formed state. We know that $\theta(\mathtt{Id_f}) = Id_H^f$ and $\theta(\mathtt{R_f}) = R_H^f$. Let us fix $i \in \mathit{Cell}$. Then, we must show that $\theta'(\mathtt{Id_f})(i) = Id_{H'}^f(i)$ and $\theta'(\mathtt{R_f})(i) = R_{H'}^f(i)$. We can prove this by a two-way case analysis over the Boolean predicate $u \in \theta(\mathtt{R_f})(i) \wedge u \neq i$.

□

5 First-Order axiomatization of well-foundedness

Consider a state $\Sigma = (\sigma, \theta)$ of the augmented program $P^{\#}$. Since the variables $\mathtt{f}$, $\mathtt{R_f}$, $\mathtt{Id_f}$ and $\mathtt{H_f}$ are not independent, we need to constrain the set of legal states Σ for the augmented program. This can be achieved by adding constraints or axioms relating the variables $\mathtt{R_f}$, $\mathtt{Id_f}$, $\mathtt{H_f}$ and $\mathtt{f}$ for any linking field $\mathtt{f}$.

In this section, we provide a set of first-order axioms that relate the variables $\mathtt{R_f}$, $\mathtt{Id_f}$, $\mathtt{H_f}$ and $\mathtt{f}$ for any linking field $\mathtt{f}$ for any well-founded state Σ. We first provide two first-order axioms (called base axioms) to capture the relationships between $\mathtt{R_f}$, $\mathtt{Id_f}$ and $\mathtt{f}$. We then provide an induction principle (*IND-WF*) that enables us to derive additional first-order axioms from the base axioms. For convenience, we will use the notation $\mathtt{f}(u)$ to denote $u.\mathtt{f}$ in this section.

5.1 Base Axioms

The following two fundamental axioms characterizes the relationship between the predicates $\mathtt{R_f}$, $\mathtt{Id_f}$, $\mathtt{H_f}$ and $\mathtt{f}$ in any state Σ of the program. In all these axioms, the cells $u \in \mathit{Cell}$, $v \in \mathit{Cell}$ are implicitly universally quantified out.

1. The first axiom specifies that $\mathtt{R_f}(u, v)$ is true if and only if either (i) there is a zero-length path from u to v (when $u = v$), or (ii) there is a path of length one or more from u to v without encountering any cells from $\mathtt{H_f}$.

$$\mathtt{R_f}(u, v) \Leftrightarrow (u = v \vee (\mathtt{f}(u) \notin \mathtt{H_f} \wedge \mathtt{R_f}(\mathtt{f(u)}, \mathtt{v}))) \quad \text{(AX1)}$$

2. The second axiom relates $\mathtt{Id_f}(u)$ with $\mathtt{Id_f}(\mathtt{f}(u))$ when $\mathtt{f}(u) \notin \mathtt{H_f}$. If $\mathtt{f}(u) \in H$, then $\mathtt{Id_f}(u) = \mathtt{f}(u)$.

$$\mathtt{Id_f}(u) = (\mathtt{f}(u) \in \mathtt{H_f})\ ?\ \mathtt{f(u)} : \mathtt{Id_f}(\mathtt{f(u)}) \quad \text{(AX2)}$$

The above axioms follow from the definition of the two predicates from Section 4.1.

We define a state Σ to be *finite* if the domain of Σ (i.e. set *Cell*) is finite in Σ. Also, recall (from Section 4.1) that for any function $f : \mathit{Cell} \rightarrow \mathit{Cell}$ and a set $H \subseteq \mathit{Cell}$, $R_H^f(u)$ defines the set of all cells $u, f(u), \ldots$ until the first cell from H is encountered, and $Id_H^f(u)$ denotes the identify of the first cell in H encountered.

The base axioms intuitively capture the definition of the fields $\mathtt{R_f}$ and $\mathtt{Id_f}$. The following theorem, which is a natural generalization of the theorem for acyclic case [21], serves to make this intuition precise.

Theorem 2 *For any finite and well-founded state Σ satisfying axioms* AX1 *and* AX2, $\Sigma(\mathtt{R_f}) = R_H^f$, *and* $\Sigma(\mathtt{Id_f}) = Id_H^f$, *where* $f = \Sigma(\mathtt{f})$ *and* $H = \Sigma(\mathtt{H_f})$.

Proof: We will sketch the proof for $\mathtt{R_f}$ and omit the proof for $\mathtt{Id_f}$, which is similar.

Let us first assume that v is reachable from u using $k \geq 0$ applications of f (i.e. $v = f^k(u)$), without encountering any cells from H. Since Σ satisfies AX1, $\Sigma(\mathtt{R_f})(f^k(u), v) = \mathit{true}$. If $k = 0$, then we are done. Otherwise, by AX1, and the fact that none of $f(u), \ldots, f^k(u)$ intersect with H, $\Sigma(\mathtt{R_f})(f^{k-1}(u), v) = \Sigma(\mathtt{R_f})(f^{k-2}(u), v) = \ldots = \Sigma(\mathtt{R_f})(u, v) = \mathit{true}$.

On the other hand, assume that $\Sigma(\mathtt{R_f})(u, v) = \mathit{true}$. Let $f^k(u)$ for $k > 0$ be the first cell in the sequence $f(u), f^2(u), \ldots,$ that is present in H. Assume that v does not equal any element of the set $\{u, f(u), f^2(u), \ldots, f^{k-1}(u)\}$. Unfolding the definition of $\mathtt{R_f}(w, v)$ $k-1$ times with $w \doteq u, w \doteq f(u), \ldots, w \doteq f^{k-1}(u)$, we get $\mathtt{R_f}(u, v) = \mathit{false}$. Hence v has to be in $\{u, f(u), \ldots, f^{k-1}(u)\}$. □

5.2 Induction Principle

Even though the axioms AX1 and AX2 capture fundamental properties of R_H^f and Id_H^f, they are not complete. As noted by Lev-Ami et al. [21], at least one of the limitation of the above axioms (and first-order logic in general) is that there is no complete axiomatization of "finiteness".

A consequence of this limitation is that above axioms might imply other derived axioms in any finite state, but these additional axioms can't be derived from the two axioms above solely by first order reasoning. These derived axioms are required to further constrain the set of states to give a meaningful assignments to $\mathtt{R_f}$, $\mathtt{Id_f}$ and $\mathtt{f}$.

In this section, we present an induction principle (*IND-WF*) that can be used to derive other theorems from the base axioms AX1 and AX2.

Definition 4 (*IND-WF) Consider a well-founded state* Σ *where* $f = \Sigma(\mathtt{f})$ *and* $H = \Sigma(\mathtt{H_f})$*, for the linking field* $\mathtt{f}$*. To show that a property* $P(u)$ *holds for all* $u \in Cell$*, it is sufficient to establish two cases:*

1. *Base Case: Establishes the property for all cells* u *such that* $f(u) \in H$*:*

$$f(u) \in H \Rightarrow P(u) \tag{1}$$

2. *Induction Step: If* P *holds for a cell* v *and* $v \notin H$*, then establish that* P *holds for all the cells* u *such that* $f(u) = v$*:*

$$(P(v) \wedge f(u) = v \wedge v \notin H) \Rightarrow P(u) \tag{2}$$

We show that if the above induction principle establishes that a predicate P is true for all the cell $u \in Cell$ in a state Σ, then Σ satisfies $\forall u : P(u)$.

Proposition 1 *The induction principle IND-WF is sound.*

Proof: Let Σ be a given state and $f = \Sigma(\mathtt{f})$ and $H = \Sigma(\mathtt{H_f})$. Let us define a metric $distance_f$ of a cell u from the cells in H:

$$distance_f(u) \quad = \quad \begin{cases} 1, & f(u) \in H \\ 1 + distance_f(f(u)), & \text{otherwise} \end{cases}$$

The base case of the induction proceeds by proving $P(u)$ for all the cells u for which $f(u) \in H$. For the induction step, we assume that the property P holds for $f(u)$, and then prove it for u. When $f(u) \notin H$, we know that the $distance_f(u) > distance_f(f(u))$ by the above definition and the well-foundedness of f. This ensures that every u has a finite and well-defined $distance_f(u)$ value. □

5.3 Derived Axioms

In this section, we present a small set of useful first-order axioms that have been required in the various proof efforts we have undertaken. These axioms can be derived from the base axioms using the induction principle described above. It is a challenge to identify a "core" subset of axioms that is not only useful in practice, but also fairly intuitive to understand. We have identified a small set of such axioms that seem to be sufficient for the set of examples handled in this paper. Of course, we can't make any claims whether this set will suffice for other programs too, since our experience is limited to the set of programs we have handled. But we believe that this set captures the interesting cases in dealing with most linked list programs.

1. **Transitivity:** The relation $\mathtt{R_f}$ enjoys the transitivity property:

$$(\mathtt{R_f}(u, v) \wedge \mathtt{R_f}(v, w)) \Rightarrow \mathtt{R_f}(u, w) \qquad (\text{ TR})$$

2. **Antisymmetry:** This property is key to breaking the symmetry in a cycle.

$$(\mathtt{R_f}(u, v) \wedge \mathtt{R_f}(v, u)) \Rightarrow u = v \qquad (\text{ AS})$$

3. **Bounded Distinctness:** For any heap cell u, if none of the cells $\mathtt{f}(u), \mathtt{f}^2(u), \ldots, \mathtt{f}^k(u)$ intersect with the set $\mathtt{H_f}$, then all the cells in this sequence are distinct from each other. Let $F_k \doteq \{\mathtt{f}(u), \ldots, \mathtt{f}^k(u)\}$, and $DISTINCT(S)$ denotes that all the members in a set S are pairwise distinct. We can derive a parameterized system of theorems DT_k for different finite values of k:

$$F_k \cap H = \{\} \Rightarrow DISTINCT(\{u\} \cup F_k) \qquad (\text{ DT}_k)$$

We have found that the case for $k = 1$ to be useful in proving properties of many linked list programs (e.g. *reverse* of an acyclic linked list):

$$(\mathtt{f}(u) \notin H) \Rightarrow u \neq \mathtt{f}(u) \qquad (\text{ DT}_1)$$

These axioms can be proved easily from the base axioms AX1 and AX2. and the induction principle *IND-WF*. We illustrate this by working through the proof of DT_1:

Proof: We will substitute the expression $(\mathtt{f}(u) \notin H) \Rightarrow u \neq \mathtt{f}(u)$, in place of $P(u)$ in Equation 1 and Equation 2. Then we derive contradiction from the negation of the formula.

- *Base Case:* Since $\mathtt{f}(u) \notin H$, the formula for the base case is unsatisfiable.
- *Induction Step:* Substituting the definition of P and rewriting v with $\mathtt{f}(u)$ in Equation 2, we get:

$$(\mathtt{f}(\mathtt{f}(u)) \in H \vee \mathtt{f}(u) \neq \mathtt{f}(\mathtt{f}(u))) \wedge$$
$$\mathtt{f}(\mathtt{f}(u)) \notin H \wedge \mathtt{f}(u) \notin H \wedge u = \mathtt{f}(u)$$

Rewriting the formula after replacing $\mathtt{f}(u)$ with u (since $u = \mathtt{f}(u)$), and removing $\mathtt{f}(u) = \mathtt{f}(u)$, we get

$$(\mathtt{f}(u) \in H \vee \mathtt{f}(u) \neq \mathtt{f}(u)) \wedge \mathtt{f}(u) \notin H$$

which is a contradiction.

□

5.4 Optimization

In addition to the instrumentation discussed in Section 4, we also introduce an `assume()` statement immediately after each statement of the form `x = y.f`:

`assume(x` $\notin$ $\mathtt{H_f}$ `=>` $\mathtt{R_f}(\mathtt{x}) = \mathtt{R_f}(\mathtt{y}) \setminus \{\mathtt{y}\}$`);`

The constraint described by the `assume()` statement is an instance of the following theorem (that can again be derived from AX1 and the induction principle):

$$\mathtt{f}(u) \notin \mathtt{H_f} \Rightarrow \mathtt{R_f}(\mathtt{f}(\mathtt{u})) = \mathtt{R_f}(\mathtt{u}) \setminus \{\mathtt{u}\} \qquad (\text{T1})$$

This can be seen as an optimization to instantiate the following derived theorem eagerly during the instrumentation phase, preventing the theorem prover to search for instances of this axiom. The axiom T1 is not added explicitly to the set of derived axioms used in the proofs.

6 Experiments

We have implemented a prototype tool to mechanically verify properties of programs containing linked lists. In this section, we briefly describe the components of the prototype and some preliminary results on a set of programs manipulating singly or doubly linked lists.

The source program is written in the Zing [1] programming language, an imperative Java-like language but without inheritance. In addition to the programming language discussed in the paper, it can also support arrays of *Boolean*, *Integer* and *Cell*.

The user specifies a subset of fields in the program as linking fields. For each field `f`, the tool automatically instruments the Zing program with the auxiliary variables $\mathtt{R_f}$, $\mathtt{Id_f}$, and adds the updates to these variables. The instrumented program also contains the asserts for null-dereference and the asserts to ensure that the fields are well-founded. The instrumented program is then translated to a guarded transition system and is fed to the UCLID verification system [7]. Let us briefly describe the UCLID system and the features that are used in this paper.

6.1 UCLID

The input language of UCLID supports variables of type Booleans, integers, and functions (and predicates) over integers. The functions can have any finite arity. Each variable of type *Cell* in the source program is mapped to an integer variable. Each field `f` is mapped to a function variable from integers to integers or Booleans. Set-valued variables such as $\mathtt{R_f}$ (respectively $\mathtt{H_f}$) are mapped to a predicate variable of arity two (respectively one). Similarly, $\mathtt{Id_f}$ is mapped to a function variable of arity one.

The updates to function and predicate variables are modeled using a λ notation used in Section 4 of this paper. The λ notation generalizes the theory of arrays, and allows us to modify an arbitrary number of entries in an array in a single step. This is convenient for expressing the updates of $\mathtt{R_f}$ and $\mathtt{Id_f}$ variables.

The user can add the necessary preconditions, loop invariants and the postconditions to check in the UCLID file. Recall that some of the asserts are automatically generated by our translator and are part of the list of properties to check in the UCLID file. The file also includes the set of base axioms and derived axioms described in Section 5.

UCLID supports specification of first-order axioms and properties of the form $\forall X : \phi(X)$, where ϕ is a quantifier-free expression over the state variables in the program. This is sufficient to express the axioms, preconditions, loop invariants, and the postconditions for all the examples that we have encountered in this work and believe is sufficient to capture the properties for most programs in practice. Set operations are modeled using quantified expressions (e.g. $\mathtt{S} = \mathtt{S_1} \cup \mathtt{S_2}$ gets translated to $\forall x : \mathtt{S}(x) \Leftrightarrow (\mathtt{S_1}(x) \vee \mathtt{S_2}(x))$.

The tool can be used in two different ways:

- **Proving verification conditions (VC)**: Given the preconditions, loop invariants and the postconditions, the tool generates a VC that is checked using the theorem prover in UCLID. The theorem prover uses quantifier instantiation to eliminate quantifiers in the formula and then uses a Boolean Satisfiability (SAT) based decision procedure for the theories of uninterpreted functions and linear arithmetic. Failed VCs result in a concrete counterexample that is immensely useful for strengthening the loop invariants, adding more preconditions or (in rare cases) adding new axioms.
- **Synthesizing loop invariants using indexed predicate abstraction**: The tool can automatically construct universally quantified invariants using simple *indexed* predicates from a set $\mathcal{P}$. Each predicate $p \in \mathcal{P}$ is a Boolean expression over the state variables and a set of *index* symbols $\mathcal{X}$ of type integers (recall *Cell* are identified as integers in UCLID). The tool then constructs the strongest loop invariant of the form $\forall \mathcal{X} : \phi(\mathcal{X})$, where $\phi(\mathcal{X})$ is a Boolean combination of the predicates in $\mathcal{P}$ [19, 12].

  For instance, given the set of predicates $\mathcal{P} = \{u = \mathtt{next}(v), v = \mathtt{prev}(u), \mathtt{R_{next}}(\mathtt{hd}, u), \mathtt{R_{next}}(\mathtt{hd}, v)\}$, with $\mathcal{X} = \{u, v\}$, the tool can compose the predicates to construct an invariant (say)

$$\forall u, v : \quad (\mathtt{R_{next}}(\mathtt{hd}, u) \wedge \mathtt{R_{next}}(\mathtt{hd}, v)) \Rightarrow (v = \mathtt{prev}(u) \Leftrightarrow u = \mathtt{next}(v))$$

  The ability to generate complex quantified invariants from simple predicates often relieves the user from writing down well-formed loop invariants. Various heuristics are also provided to often infer most predicates [20].

In the next two subsections, we describe our initial experience with using the two features of UCLID to verify a set of programs manipulating linked lists.

6.2 Benchmarks

We have currently handled the following set of examples using at least one of the two tools mentioned above:

- `acyclic_simple, cyclic_simple`: These are the examples from Section 1 where all the elements of the list are initialized to 0.
- `reverse_a`: This example performs an in-place destructive update of an acyclic linked list pointed to by a variable `l`.
- `setunion, setunion_S1`: This is the example described in Figure 1. The example `setunion_S1` denotes the portion of the example before the destructive updates to combine the two lists into one. We have included both `setunion` and `setunion_S1` to demonstrate the increase in complexity in the presence of destructive updates.
- `insert`: This is the example of insertion into a sorted list described in Figure 2 in Section 2.
- `dlist_remove`: This example of a doubly linked list is also described in Section 2.

6.3 Proving verification conditions

In this section, we describe the set of examples for which we provided the loop invariant manually and used the theorem prover to prove the verification conditions.

Figure 5 describe the results of verifying these examples on a 2 GHz machine running Linux and 1GB memory. ZChaff [26] was used as the SAT solver inside UCLID. We have plotted the complexity of the VC generated (VC Size)

Example	VC Size	# Instants	Time Taken in UCLID (sec)
`cyclic_simple`	991 (45)	0	2.1
`reverse_a`	1896 (72)	0	9.5
`setunion_S1`	1134 (49)	0	3.9
`setunion`	2960 (104)	1	11.4
`insert`	5950(110)	2	37.5
`dlist_remove`	3362 (115)	1	30.1

Figure 5: Results of verifying linked list programs. "VC Size" denotes the number of nodes in the VC formula after instantiation; the numbers in the parenthesis denote the number of integer-valued terms in the formula. The column "# Instants" denotes the number of manual instantiations that had to be done.

Example	# Predicates	# Iterations	Time Taken (sec)
`cyclic_simple`	15(1)	4	11.4
`reverse_a`	16(1)	6	85.45
`set_union`	24 (1)	5	79.79
`insert`	21(2)	9	1404.07

Figure 6: Results of verifying linked list programs using indexed predicate abstraction. "# Predicates" denotes the number of predicates requires; the number in the parenthesis denotes the number of index variables in the predicates. "# Iterations" is the number of iterations of abstract reachability computation to compute the invariant. "Time taken" is the time taken to construct the invariant and prove the postconditions.

and the total time taken by the theorem prover to prove the formula. We have also included the number of cases where we had to manually instantiate an axiom.

Observe the difference between the verification of `setunion_S1` and `setunion`. In the latter case, adding the destructive updates requires one manual instantiation as well as increased CPU time. Also, the complexity of verifying `dlist_remove` comes from two sources even though the program is just four lines long: first, the precondition of the method is complex, and secondly, the set of axioms double and number terms to instantiate increases because of the presence of two fields (`next`, `prev`) in the program.

The manual instantiation of quantifiers in the axioms have come mainly from two cases so far:

- The quantifier instantiation heuristics fail to infer relevant terms to instantiate. It only happened once for the example `insert`, where we had to instantiate AX2 with a concrete term.
- Since the axiom TR relates three bound variables u, v and w, and the number of combinations to instantiate grows exponentially with the number bound variables resulting in a very large VC, we always use the axiom after instantiating the first argument u with one of the program variables. This was the source of one instantiation for each of `setunion`, `insert` and `dlist_remove` examples.

Both the above problems can be mitigated by better quantifier instantiation strategies, and we are working towards improving the heuristics in UCLID. We do not use Simplify to discharge the VCs because the concrete counterexample facility in UCLID offsets the value of more advanced instantiation heuristics present in Simplify.

6.4 Invariant synthesis

We have also leveraged the predicate abstraction engine in UCLID to infer loop invariants for some of the examples. Our initial experience has been encouraging, and we report some preliminary results in this section. At present, we have managed to construct the loop invariants for the `cyclic_simple`, `reverse_a`, `set_union` and `insert` examples, given a set of indexed predicates.

Figure 6 illustrates the result of synthesizing loop invariants using indexed predicate abstraction for a subset of the examples. Currently the tool suffers from two bottlenecks that results in significant time to construct the invariants. This also explains the time taken to prove the verification conditions in the last section.

- UCLID does not maintain the control flow graph explicitly and encodes the program counter as a variable — the entire program is encoded as a single transition relation. This is because the tool was primarily built for analyzing distributed protocols and systems. This results in a large blowup in the formulas that the theorem prover or the predicate abstraction engine gets, and consequently slows the analysis. We are currently working on incorporating explicit control flow into the tool.
- The quantifier instantiation engine generates a large number of (often redundant) terms to instantiate. Moreover, the number of combinations to instantiate grows exponentially with the number of index variables in $\mathcal{X}$. For instance, with two index variables, the number of combinations to instantiate went up to 81 for the `insert` example. In some cases, SAT often saw formulas with more than 300K clauses in them. To mitigate this problem, we are exploring alternate quantifier instantiation and predicate abstraction strategies. Besides, one can often trade off the precision of the predicate abstraction to construct less precise loop invariants more efficiently. We plan to investigate if such loop invariants suffice to prove the properties of interest for these candidate programs.

Figure 7 describe the `reverse_a` example along with the set of predicates required to prove the postcondition. The example required a single index symbol (denoting a heap cell) u to construct the loop invariant. The tool requires 25.03 seconds to construct the loop invariant and prove the property. The set of predicates for this example was supplied manually. We are currently working on automating the process of predicate discovery to make the verification more automated.

7 Related Work

There is a rich body of work in reasoning about programs that perform destructive updates of heap allocated data structures. Work in this area can be divided into the following often overlapping categories: (1) first-order axiomatization of reachability, (2) shape analysis, (3) using decidable

```
//@ requires null ∈ H_next
//@ requires Id_next(l) == null

//@ ensures R_next(res) == old(R_next)(l)

Cell reverse (Cell l) {
     Cell curr = l;
     Cell res = null;

     while (curr != null) {
          Cell tmp = curr.next;
          curr.next = res;
          res = curr;
          curr = tmp;
     }

     return res;
}
```

Set of Predicates
$\mathcal{X} = \{u\}$
$\mathcal{P} = \{u = null, u = \texttt{curr}, u = \texttt{res}, u = \texttt{old(l)},$
$\texttt{R}_{\texttt{next}}(\texttt{curr}, u), \texttt{R}_{\texttt{next}}(\texttt{res}, u), \texttt{H}_{\texttt{next}}(\texttt{u}),$
$\texttt{old}(\texttt{R}_{\texttt{next}})(\texttt{old(l)}, u), \texttt{l} = \texttt{old(l)}, \texttt{R}_{\texttt{next}}(\texttt{old(l)}, u),$
$\texttt{curr} = null, \texttt{Id}_{\texttt{next}}(u) = null\}$

Figure 7: Reversing an acyclic list. The source program along with the set of predicates. We use `old(x)` to denote the value of `x` at the method entry.

logics to capture reachability information, (4) using predicate abstraction to construct inductive invariants for linked lists, and (5) local reasoning using separation logic.

First-order axiomatization of reachability. The work in this category is closest to our work. Nelson [28] proposed the *ternary* reachability predicate $u \xrightarrow[x]{f} v$ to define that u can reach v through applications of f without encountering x, and provided a set of first-order axioms to capture this predicate. Our axiomatization is based on two *binary* predicates. We believe that our predicates are more intuitive to a programmer and our axiomatization yields simpler correctness proofs. For example, Nelson's proof of the set-union example from Section 2.1 required eight axioms whereas our proof requires only the transitivity axiom in addition to the two base axioms. Lev-Ami et al. [21] proposed another set of axioms for characterizing the reachability predicate. Their approach works only for acyclic lists and will not allow them to express loop invariants for programs manipulating cyclic lists. McPeak [24] proposes a methodology for writing specifications of data structures in first-order logic. Since he does not attempt to axiomatize reachability, his method, by itself, is not sufficient to express reachability or disjointness properties of linked lists. To express such facts, the programmer has to manually introduce and update ghost variables in a program specific way. His work, however, provides new heuristics for quantifier instantiation targeted towards reasoning for linked data structures. This work should complement our work which depends crucially on first-order theorem provers.

Shape analysis. Work in this category attempts to reason about the shape properties of the heap, such as acyclicity and sharing, in the presence of destructive updates to the heap. Ghiya and Hendren [13] propose the use of two boolean matrices to record if two pointers are reachable from each other and if they share any heap cell. They provide conservative updates to these matrices for various statements in the program. Sagiv et al. [32, 33] use a 3-valued logic to represent abstractions of the heap graph. For improved precision, their approach requires *instrumentation predicates*, which usually use a reachability predicate as a building block. The updates for these predicates are either supplied by the user or constructed conservatively using a theorem prover that can reason about transitive closure. The idea is implemented in the TVLA [22] system and has been used to infer loop invariants for various linked data structures. In contrast to this work, our approach depends on purely first-order reasoning. Recently, Hackett and Rugina [15] have proposed a method that uses points-to information to construct an abstraction of the heap. Their approach cannot encode relationships (e.g. equality) between program expressions and cannot describe doubly-linked lists. One advantage of our approach compared to all of the above is the ease with which we can combine reasoning about linked lists with reasoning about arithmetic and arrays.

Decidable logics. Various decidable logic fragments have been also proposed to express properties of linked data structures. PALE [25] uses monadic second-order logic to express properties of lists, trees, and graphs. The logic L_r [5] was proposed to reason about reachability. However, the decidability results quickly break down in the presence of complex shapes and scalar values in the program.

Predicate abstraction for discovering invariants. Recently, predicate abstraction [14] has been extended to construct invariants for linked-list programs. Dams and Namjoshi [8] proposed the use of reachability predicate to construct an abstraction of the concrete system. They provide a heuristic for predicate discovery based on constructing the weakest precondition of the reachability predicate. Manevich et al. [23] observe that the number of shared nodes in the heap consisting of singly-linked lists is statically bounded, and propose a family of predicates to exploit this observation. The idea has been implemented in TVLA. Unlike our work, both the above methods require a theorem prover that can reason about transitive closure to construct the abstraction. Balaban et al. [2] provide a decision procedure for restricted formulae involving the reachability predicate and provide a method to compute the abstraction. The restrictions on the reachability predicate does not allow them, for example, to express the loop invariant for the `cyclic_simple` program from Section 1.1. In comparison to all of the above, our ability to harness invariant inference methods for first-order logic (e.g. indexed predicate abstraction) provides us with appreciable automation for programs that manipulate linked structures, arrays, and other scalar values.

Separation logic. Separation logic [18, 31] is a promising idea for local reasoning of heap-manipulating programs. Separation logic naturally extends traditional Hoare-style reasoning to bear upon such programs. However, most of the work on separation logic focused on proving programs manually [29, 30]. Recently, Berdine et al. [6] have developed decision procedures for fragments of separation logic. However, we are not aware of any automated tools for program verification based on these decision procedures. Proofs

of linked-list programs using separation logic use the reachability predicate; our axiomatization of reachability could potentially help with mechanizing such proofs. Alternatively, our work can benefit from separation logic specifications in future as we extend our work to the inter-procedural setting.

8 Conclusions

Programs such as kernels of operating systems and device drivers manipulate a variety of data structures, such as arrays, singly and doubly linked lists, and hashtables. Current verification tools focus on control-dominated properties and are consequently unable to handle such programs in which the control flow interacts with the data in subtle ways. This paper is our first step towards the goal of building a scalable checker for verifying low-level data-rich systems software.

In this paper, we presented a novel method for verifying linked data structures based on a first-order axiomatization of reachability with respect to a set of head cells. This axiomatization is based on the idea of a well-founded heap and has the advantage that acyclic and cyclic lists are handled uniformly with equal ease. We have implemented our method in a tool and used it to verify the correctness of a variety of nontrivial programs manipulating both acyclic and cyclic singly-linked lists and doubly-linked lists.

There are several immediate directions for future work. We would like to verify more examples that use doubly-linked lists and arrays to evaluate the overhead of using our methodology and the adequacy of our list of derived first-order axioms. We have access to several programs from the Windows kernel that use rich data structures; we intend to evaluate our method on these programs. We would like to automate the inference of invariants more by developing heuristics for predicate discovery targeted towards linked lists. Finally, we would like to extend our work to deal with procedure calls and develop techniques to summarize procedures.

Acknowledgements

We are grateful to Thomas Ball for his insightful comments on this paper.

References

[1] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR 2004: 15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2004.

[2] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis by predicate abstraction. In *VMCAI 05: Verification, Model checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 164–180. Springer-Verlag, 2005.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213, 2001.

[4] C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Computer Aided Verification (CAV '04)*, LNCS 3114. Springer-Verlag, 2004.

[5] Michael Benedikt, Thomas W. Reps, and Shmuel Sagiv. A decidable logic for describing linked data structures. In *ESOP 99: European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 2–19. Springer-Verlag, 1999.

[6] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS 04: Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, 2004.

[7] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.

[8] Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI 03: Verification, Model checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 310–324. Springer-Verlag, 2003.

[9] D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical report, HPL-2003-148, 2003.

[10] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.

[12] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Symposium on Principles of programming languages (POPL '02)*, pages 191–202. ACM Press, 2002.

[13] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL 96: Principles of Programming Languages*, pages 1–15, 1996.

[14] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*, LNCS 1254. Springer-Verlag, June 1997.

[15] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *POPL 05: Principles of Programming Languages*, pages 310–323, 2005.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Symposium on Principles of programming languages (POPL '02)*, pages 58–70. ACM Press, 2002.

[17] Neil Immerman, Alexander Moshe Rabinovich, Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *18th International Workshop on Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 160–174. Springer-Verlag, 2004.

[18] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL 01: Principles of Programming Languages*, pages 14–26, 2001.

[19] S. K. Lahiri and R. E. Bryant. Constructing Quantified Invariants via Predicate Abstraction. In *Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, LNCS 2937, pages 267–281, 2004.

[20] S. K. Lahiri and R. E. Bryant. Indexed Predicate Discovery for Unbounded System Verification. In *Computer Aided Verification (CAV '04)*, LNCS 3114. Springer-Verlag, 2004.

[21] Tal Lev-Ami, Neil Immerman, Thomas W. Reps, Shmuel Sagiv, Siddharth Srivastava, and Greta Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *20th Conference on Automated Deduction (CADE '05) (to appear)*, 2005.

[22] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *SAS 00: Static Analysis Symposium*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer-Verlag, 2000.

[23] Roman Manevich, Eran Yahav, G. Ramalingam, and Mooly Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*, volume 3148 of *Lecture Notes in Computer Science*, pages 181–198. Springer-Verlag, 2005.

[24] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *Computer-Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 476–490. Springer-Verlag, 2005.

[25] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI 01: Programming Language Design and Implementation*, pages 221–231, 2001.

[26] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC '01)*, 2001.

[27] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):245–257, 1979.

[28] Greg Nelson. Verifying reachability invariants of linked structures. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 38–47. ACM Press, 1983.

[29] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL 01: 10th International Workshop on Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001.

[30] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL 04: Principles of Programming Languages*, pages 268–280, 2004.

[31] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 02: Logic in Computer Science*, pages 55–74, 2002.

[32] R. Wilhelm S. Sagiv, T. W. Reps. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):1–50, 1998.

[33] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL 99: Principles of Programming Languages*, pages 105–118. ACM, 1999.