Automatic Induction Proofs of Data-Structures in Imperative Programs

Duc-Hiep Chu
National University of Singapore
Republic of Singapore
hiepcd@comp.nus.edu.sg

Joxan Jaffar
National University of Singapore
Republic of Singapore
joxan@comp.nus.edu.sg

Minh-Thai Trinh
National University of Singapore
Republic of Singapore
trinhmt@comp.nus.edu.sg

Abstract

We consider the problem of automated reasoning about dynamically manipulated data structures. Essential properties are encoded as predicates whose definitions are formalized via user-defined recursive rules. Traditionally, proving relationships between such properties is limited to the *unfold-and-match* (U+M) paradigm which employs systematic transformation steps of folding/unfolding the rules. A proof, using U+M, succeeds when we find a sequence of transformations that produces a final formula which is obviously provable by simply *matching* terms.

Our contribution here is the addition of the fundamental principle of *induction* to this automated process. We first show that some proof obligations that are dynamically generated in the process can be used as *induction hypotheses* in the future, and then we show how to use these hypotheses in an *induction step* which generates a new proof obligation aside from those obtained by using the fold/unfold operations. While the adding of induction is an obvious need in general, no automated method has managed to include this in a systematic and general way. The main reason for this is the problem of avoiding *circular reasoning*. We overcome this with a novel checking condition. In summary, our contribution is a proof method which – beyond U+M – performs *automatic* formula re-writing by treating previously encountered obligations in each proof path as possible induction hypotheses.

In the practical evaluation part of this paper, we show how the commonly used technique of using unproven *lemmas* can be avoided, using realistic benchmarks. This not only removes the current burden of coming up with the appropriate lemmas, but also significantly boosts up the verification process, since lemma applications, coupled with unfolding, often induce a large search space. In the end, our method can automatically reason about a new class of formulas arising from practical program verification.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods; D.2.4 [*Software/Program Verification*]: Correctness proofs; F.4.1 [*Mathematical Logic*]: Proof theory

General Terms Verification

Keywords Data-structures, Induction Proofs, Separation.

PLDI'15, June 13–17, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3468-6/15/06...$15.00
http://dx.doi.org/10.1145/2737924.2737984

1. Introduction

We consider the automated verification of imperative programs with emphasis on reasoning about the functional correctness of dynamically manipulated data structures. The dynamically modified heap poses a big challenge for logical methods. This is because typical correctness properties often require combinations of structure, data, and *separation*.

Automated proofs of data structure properties — usually formalized using Separation Logic (or the alike) and extended with *user-defined* recursive predicates — "rely on decidable sub-classes together with the corresponding proof systems based on (un)folding strategies for recursive definitions" [24]. Informally, in the regard of handling recursive predicates, the state-of-the-art [9, 22, 26–28], to name a few, collectively called unfold-and-match (U+M) paradigm, employ the basic but systematic transformation steps of *folding* and *unfolding* the rules.

A proof, using U+M, succeeds when we find successive applications of these transformation steps that produce a final formula which is *obviously* provable. This usually means that either (1) there is no recursive predicate in the RHS of the proof obligation and a direct proof can be achieved by consulting some generic SMT solver; or (2) no special consideration is needed on any occurrence of a predicate appearing in the final formula. For example, if $\mathtt{p}(\tilde{u}) \wedge \cdots \models \mathtt{p}(\tilde{v})$ is the formula, then this is obviously provable if $\tilde{u}$ and $\tilde{v}$ were *unifiable* (under an appropriate theory governing the meaning of the expressions $\tilde{u}$ and $\tilde{v}$). In other words, we have performed "formula abstraction" [22] by treating the recursively defined term $\mathtt{p}()$ as *uninterpreted*.

A key feature that is missing from the U+M methodology is the ability to prove by *induction*, which is often required in verification of practical examples [2]. Without inductive reasoning, U+M (folding/unfolding together with formula abstraction) *cannot* handle proof obligations involving *unmatchable* predicates. Specifically, in such obligations, there exists a recursively defined predicate in the RHS which cannot be transformed, via folding/unfolding, to one that is unifiable with some predicate in the LHS.

As a concrete example, consider the following definitions of list and list of zero numbers:

$$\begin{aligned} \mathsf{vlist}(\mathtt{x}) &\stackrel{def}{=} \mathtt{x}{=}\mathtt{null} \wedge \mathbf{emp} \\ &\mid (\mathtt{x}{\mapsto}\_, \mathtt{t}) \star \mathsf{vlist}(\mathtt{t}) \\ \mathsf{zero\_list}(\mathtt{x}) &\stackrel{def}{=} \mathtt{x}{=}\mathtt{null} \wedge \mathbf{emp} \\ &\mid (\mathtt{x}{\mapsto}0, \mathtt{t}) \star \mathsf{zero\_list}(\mathtt{t}) \end{aligned}$$

In Fig. 1, we present a partial proof that a list of zero elements is a list. First, by unfolding the LHS, the original proof obligation is resolved into (i) and (ii). The first sub-obligation can be easily discharged by unfolding the RHS. (It is clear that U+M is inadequate for this proof. This is because no matter how we apply folding/unfolding, there

$$\textsc{(left-unfold)}\ \dfrac{\textsc{(right-unfold)}\ \dfrac{\textsc{(obvious)}\ \dfrac{\texttt{True}}{\texttt{x=null}\wedge\mathbf{emp}\models\texttt{x=null}\wedge\mathbf{emp}}}{\texttt{x=null}\wedge\mathbf{emp}\models\mathsf{vlist}(\texttt{x})\ \text{(i)}}\qquad(\texttt{x}\mapsto 0,\texttt{t})*\mathsf{zero\_list}(\texttt{t})\models\mathsf{vlist}(\texttt{x})\ \text{(ii)}}{\mathsf{zero\_list}(\texttt{x})\models\mathsf{vlist}(\texttt{x})}$$

Figure 1: Partial Proof Tree for $\mathsf{zero\_list}(\texttt{x}) \models \mathsf{vlist}(\texttt{x})$

$$\textsc{(left-unfold)}\ \dfrac{\textsc{(right-unfold)}\ \dfrac{\textsc{(obvious)}\ \dfrac{\texttt{True}}{\texttt{x=null}\wedge\mathbf{emp}\models\texttt{x=null}\wedge\mathbf{emp}}}{\texttt{x=null}\wedge\mathbf{emp}\models\mathsf{zero\_list}(\texttt{x})\ (1)}\qquad(\texttt{x}\mapsto\_,\texttt{t})*\mathsf{vlist}(\texttt{t})\models\mathsf{zero\_list}(\texttt{x})\ (2)}{\mathsf{vlist}(\texttt{x})\models\mathsf{zero\_list}(\texttt{x})}$$

Figure 2: Partial Proof Tree for $\mathsf{vlist}(\texttt{x}) \models \mathsf{zero\_list}(\texttt{x})$

still exists a predicate vlist in the RHS, which cannot be matched with the predicate zero_list in the LHS.)

Now let us consider the original proof obligation $\mathsf{zero\_list}(\texttt{x}) \models \mathsf{vlist}(\texttt{x})$ as an *induction hypothesis*. This justifies an *induction step* comprising a transformation of (ii) into a simpler obligation, as follows: weaken the LHS by replacing $\mathsf{zero\_list}(\texttt{t})$ with $\mathsf{vlist}(\texttt{t})$, and obtain the new proof obligation (iii). It is now easy to prove (iii) by unfolding the RHS, followed by substituting $\texttt{z}$ by $\texttt{t}$. All the above steps are summarized below, where LEFT-WEAKEN denotes the transformation above.

$$\textsc{(left-weaken)}\ \dfrac{\textsc{(right-unfold)}\ \dfrac{\textsc{(substitution)}\ \dfrac{\textsc{(obvious)}\ \dfrac{\texttt{True}}{(\texttt{x}\mapsto 0,\texttt{t})*\mathsf{vlist}(\texttt{t})\models(\texttt{x}\mapsto 0,\texttt{t})*\mathsf{vlist}(\texttt{t})}}{(\texttt{x}\mapsto 0,\texttt{t})*\mathsf{vlist}(\texttt{t})\models(\texttt{x}\mapsto 0,\texttt{z})*\mathsf{vlist}(\texttt{z})}}{(\texttt{x}\mapsto 0,\texttt{t})*\mathsf{vlist}(\texttt{t})\models\mathsf{vlist}(\texttt{x})\ \text{(iii)}}}{(\texttt{x}\mapsto 0,\texttt{t})*\mathsf{zero\_list}(\texttt{t})\models\mathsf{vlist}(\texttt{x})\ \text{(ii)}}$$

While the usefulness of having such a step is very clear, the conditions for its *correct* application is not obvious. To see this, let us use the same approach now but to prove that a list is also a list of zero elements, something that is clearly false. See Fig. 2. We proceed similarly as in the previous proof:

$$\textsc{(right-strengthen)}\ \dfrac{\textsc{(right-unfold)}\ \dfrac{\textsc{(substitution)}\ \dfrac{\textsc{(obvious)}\ \dfrac{\texttt{True}}{(\texttt{x}\mapsto\_,\texttt{t})*\mathsf{vlist}(\texttt{t})\models(\texttt{x}\mapsto\_,\texttt{t})*\mathsf{vlist}(\texttt{t})}}{(\texttt{x}\mapsto\_,\texttt{t})*\mathsf{vlist}(\texttt{t})\models(\texttt{x}\mapsto\_,\texttt{z})*\mathsf{vlist}(\texttt{z})}}{(\texttt{x}\mapsto\_,\texttt{t})*\mathsf{vlist}(\texttt{t})\models\mathsf{vlist}(\texttt{x})\ (3)}}{(\texttt{x}\mapsto\_,\texttt{t})*\mathsf{vlist}(\texttt{t})\models\mathsf{zero\_list}(\texttt{x})\ (2)}$$

Once again, we use the original proof obligation $\mathsf{vlist}(\texttt{t}) \models \mathsf{zero\_list}(\texttt{t})$ as an induction hypothesis, and this time, we transform the proof obligation (2) into (3): strengthen the RHS by replacing $\mathsf{zero\_list}(\texttt{x})$ with $\mathsf{vlist}(\texttt{x})$. Call this transformation RIGHT-STRENGTHEN. Clearly (3) is easily proven true, as shown.

This erroneous proof arises from a form of *circular reasoning*. Our challenge therefore is how to use induction correctly, as in Fig. 1, but avoid pitfalls such as in Fig. 2.

In this paper, we propose a general proof method for recursive predicates that includes reasoning by induction. Our method is able to use *dynamically generated* formulas as induction hypotheses, and to enforce an *anti-circular* condition so that any application of an induction step is guaranteed to be correct. We shall see that our method is very different from that in traditional theorem proving systems where, after having chosen an induction tactic, the system will then search for appropriate induction variable(s) with a well-founded measure and appropriate induction hypotheses. In our framework, the predicates are defined by general recursive rules, without any explicit restriction to any well-founded orderings, and includes a domain of discourse that captures the mutable heap and properties of separation. More specifically:

- We automatically and efficiently *discharge* all commonly-used lemmas, extracted from a number of benchmarks used by other systems. These systems cannot automatically discharge such lemmas, but simply accept them as true facts.
- We demonstrate, in a different set of benchmarks in Section 5, that with our proof method, the common usage of lemmas can be *avoided*. This is because the properties of interest are covered by our method. In contrast, these properties cannot be discharged by the other systems without using lemmas.

  The impact of this is twofold. First, it means that for proving practical (but small) programs, the users are now free from the burden of providing custom user-defined lemmas. Second, it significantly boosts up the performance, since lemma applications, coupled with folding/unfolding, often induce a large search space.
- The proposed proof method gets us back the power of compositional reasoning in dealing with user-defined recursive predicates. While we have not been able to identify *precisely* the class where our proof method would be effective[1]; we do believe that its potential impact is huge. One important subclass that we can handle effectively is when both the antecedent and the consequent refer to the same structural shape but the antecedent simply makes a *stronger* statement about the values in the structure (e.g., to prove that a sorted list is also a list, an AVL tree is also a binary search tree, a list consists of all data values 999 is one that has all positive data, etc.).

In summary, we extend significantly the state-of-the-art proof methods, namely U+M based methods. We are able to prove relationships between general predicates of arbitrary arity, even when recursive definitions and the code are structurally dissimilar. In Section 2, we will motivate the need for our extension in more detail. Sections 3 and 4 contain the technical core. In Section 5, we evaluated our prototype implementation on a comprehensive set of benchmarks, including both academic algorithms and real programs. The benchmarks are collected from existing systems [8, 9, 22, 25, 28], those considered as the state-of-the-art for the purpose of proving user-defined recursive data-structure properties in imperative languages. Section 6 discusses related work in detail and Section 7 concludes.

2. Motivation

In this Section, we motivate the need for inductive reasoning in proving user-defined recursive data-structure properties.

We first highlight scenarios, which are *ubiquitous* in realistic programs, and often lead to proof obligations involving unmatchable

[1] This is as *hard* as identifying the class where an invariant discovery technique guarantees to work.

predicates. Later, we discuss the restriction of U+M paradigm in dealing with such proof obligations.

2.1 Scenario 1: Recursion Divergence

> when the "recursion" in the recursive rules is structurally dissimilar to the program code.

This happens often with *iterative* programs and when the predicates are not *unary*, i.e., they relate two or more pointer variables, from which the program code traverse/manipulate the data structure in directions different from the definition.

```
elm = malloc()          assume(head!=null)
assume(tail!=null)      assume(head!=tail)
elm.next = null         elm = head
tail.next = elm         head = head.next
tail = elm              free(elm)
```

(a) Insert Tail (b) Remove Head

$$\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) \overset{def}{=} \mathtt{x}{=}\mathtt{y} \wedge \mathbf{emp} \mid \mathtt{x}{\neq}\mathtt{y} \wedge (\mathtt{x}{\mapsto}\mathtt{t}) * \widehat{\mathsf{ls}}(\mathtt{t},\mathtt{y})$$

(c) List Segment Definition

Figure 3: Implementation of a Queue

To illustrate, Fig. 3 shows the implementation of a queue using list segment, extracted from `OpenBSD/queue.h`, an open source program. Two operations of interest: (1) adding a new element into the end of a non-empty queue (`enqueue`, Fig. 3(a)); (2) deleting an element at the beginning of a non-empty queue (`dequeue`, Fig. 3(b)). A simple property we want to prove is that given a list segment representing a non-empty queue at the beginning, after each operation, we still get back a list segment.

In the two use cases, the "moving pointers" are necessary to recurse differently: the `tail` is moved in `enqueue` while the `head` is moved in `dequeue`. Consequently, no matter how we define list segments[2], where `head` and `tail` are the two pointers, at least one use case would recurse differently from the definition, thus exhibit the "recursion divergence" scenario and lead to a proof obligation involving unmatchable predicates. More concretely, if list segment is defined as in Fig. 3(c), the `enqueue` operation would lead to an obligation that is impossible for U+M to prove.

2.2 Scenario 2: Generalization of Predicate

> when the predicate describing a loop invariant or a function needs to be used later to prove a weaker property.

This happens in almost all realistic programs. The reason is because verification of functional correctness is performed *modularly*. More specifically, given the specifications for functions and invariants for loops, we can first perform *local* reasoning before composing the whole proof for the program using, in the context of Separation Logic, the *frame rule* [32]. It can be seen that, given such divide-and-conquer strategy, at the *boundaries* between local code fragments, we would need "generalization of predicate". A particularly important relationship between predicates, at the boundary point, is simply that one (the consequent) is *more general* than the other (the antecedent), representing a valid abstraction step.

Consider the boundaries between function calls, illustrated by the pattern in Fig. 4(a). We start with the pre-condition Φ, calling function `func_a` and then `func_b`. We then need to establish the post-condition Ψ. In traditional forward reasoning, we will write local (and consistent) specifications for `func_a` and `func_b` such that: (1) Φ is stronger than the pre-condition of `func_a`; (2)

[2] Typically, list segment can be defined in two ways: the moving pointer is either the *left* one or the *right* one.

pre-condition: Φ
```
 func_a()
 func_b()
```
post-condition: Ψ

(a) Multiple Function Calls

pre-condition: Φ
` loop:` *invariant* `I`
post-condition: Ψ

(b) Iterative Loops

Figure 4: Modular Program Reasoning

the post-condition of `func_a` is stronger than the pre-condition of `func_b`; (3) the post-condition of `func_b` is stronger than Ψ. It is hard, if not impossible, to ensure that for each pair (out of three) identified above, the antecedent and the consequent are constructed from matchable predicates. As a concrete example, in `bubblesort` program [9], a boundary between two function calls requires us to prove that a *sorted* linked-list is also a linked-list.

We further argue that in software development, code reuse is often desired. The specification of a function, especially when it is a *library* function, should (or must) be relatively *independent* of the context where the function is plugged in. In each context, we might want to establish arbitrarily different properties, as long as they are *weaker* than what the function can guarantee. In such cases, it is almost certain that we will have proof obligations involving unmatchable predicates.

Now consider the boundaries caused by loops. In *iterative* algorithms, the loop invariants must be consistent with the code, and yet these invariants are only used later to prove a property often *not* specified using the identical predicates of the invariants. In the pattern shown by Fig. 4(b), this means that the proof obligations relating the pre-condition Φ to the invariant `I` and `I` to post-condition Ψ often involve unmatchable predicates. For example, programs manipulate lists usually have loops of which the invariants need to talk about list segments. Assume that (acyclic) linked-list is defined as below:

$$\mathsf{list}(\mathtt{x}) \overset{def}{=} \mathtt{x}{=}\mathtt{null} \wedge \mathbf{emp} \mid (\mathtt{x}{\mapsto}\mathtt{t}) * \mathsf{list}(\mathtt{t})$$

Though $\widehat{\mathsf{ls}}$ and list are closely related, U+M can prove neither of the following obligations:

$$\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{null}) \models \mathsf{list}(\mathtt{x}) \tag{2.1}$$

$$\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) * \mathsf{list}(\mathtt{y}) \models \mathsf{list}(\mathtt{x}) \tag{2.2}$$

In summary, the above discussion connects to a serious issue in software development and verification: without the ability to relate predicates — when they are unmatchable — *compositional reasoning* is seriously hampered.

2.3 On Unfold-and-Match (U+M) Paradigm

As stated in Section 1, the dominating technique to manipulate user-defined recursive predicates is to employ the basic transformation steps of folding and unfolding the rules, together with formula abstraction, i.e., the U+M paradigm.

The main challenge of the U+M paradigm is clearly how to systematically search for such sequences of fold/unfold transformations. We believe recent works [22, 28], we shall call the DRYAD works, have brought the U+M to a new level of automation. The key technical step is to use the *program statements* in order to *guide* the sequence of fold/unfold steps of the recursive rules which define the predicates of interest. For example, assume the definition for list segment $\widehat{\mathsf{ls}}$ in Fig. 3(c) and the code fragment in Fig. 5(a).

$\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y})$
```
 assume(x != null)
 z = x.next
```
$\widehat{\mathsf{ls}}(\mathtt{z},\mathtt{y})$

(a) Code Fragment 1

$\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) * (\mathtt{y}{\mapsto}\_)$
```
 z = y.next
```
$\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{z})$

(b) Code Fragment 2

Figure 5: U+M with List Segments

Here we want to prove that given $\widehat{\mathsf{ls}}$`(x,y)` at the beginning, we should have $\widehat{\mathsf{ls}}$`(z,y)` at the end. Since the code touches the "footprint" of `x` (second statement), it *directs* the unfolding of the predicate $\widehat{\mathsf{ls}}$`(x,y)` containing `x`, to expose $\mathtt{x} \neq \mathtt{y} \wedge (\mathtt{x}{\mapsto}\mathtt{t}) * \widehat{\mathsf{ls}}(\mathtt{t},\mathtt{y})$. The consequent can then be established via a simple *matching* from variable `z` to `t`.

Now we consider the code fragment in Fig. 5(b): instead of moving one position away from `x`, we move one away from `y`. To be convinced that U+M, however, cannot work, it suffices to see that unfolding/folding of $\widehat{\mathsf{ls}}$ does not change the *second argument* of the predicate $\widehat{\mathsf{ls}}$. Therefore, regardless of the unfolding/folding sequence, the arguments `y` on the LHS and `z` on the RHS would maintain and can never be matched satisfactorily.

The example in Fig. 5(b) exhibits the "recursion divergence" scenario mentioned above and ultimately is about relating two possible definitions of list segment (recursing either on the left or on the right pointer), which U+M fundamentally cannot handle. We will revisit this example in later Sections.

On Using Axioms and Lemmas: For systems that support general user-defined predicates [9, 28], they get around the limitation of U+M via the use, without proof, of additional *user-provided* "lemmas" (the corresponding term used in [28] is "axioms"). As a matter of fact, in the viewpoint of proof method, it is unacceptable that in order to prove more programs, we *continually* add in more custom lemmas to facilitate the proof system.

3. The Assertion Language $CLP(\mathcal{H})$

The explicit naming of heaps has emerged naturally in several extensions of Separation Logic (SL) as an aid to practical program verification. Reynolds conjectured that referring explicitly to the current heap in specifications would allow better handles on data structures with sharing [32]. In this vein, [13] extends Hoare Logic with explicit heaps. This extension allows for strongest post conditions, and is therefore suitable for "practical program verification" [6] via constraint-based symbolic execution.

In this paper, we start with the existing specification language in [13], which has two notable features: (a) the use of explicit heap variables, and (b) user-defined recursive properties in a wrapper logic language based on recursive rules. The language provides a new level of expressiveness for specifying properties of heap-manipulating programs. We remark that, common specifications written in traditional Separation Logic, can be automatically compiled into this language.

Due to space limit, we will be brief here and refer interested readers to [13] for more details. A *heap* is a *finite partial map* from *positive* integers to integers, i.e., $\mathsf{Heaps} = \mathbb{Z}_+ \rightharpoonup_{\mathsf{fin}} \mathbb{Z}$. Given a heap $h \in \mathsf{Heaps}$ with domain $D = dom(h)$, we sometimes treat h as the set of pairs $\{(p, v) \mid p \in D \wedge v = h(p)\}$. We note that when a pair (p, v) belongs to some heap h, it is necessary that p is not `null` ($p \neq 0$). The $\mathcal{H}$-language is the first-order language over heaps.

We use $(*)$ and $(\simeq)$ operators to respectively denote heap disjointness and equation. Intuitively, a constraint like $H \simeq H_1 * H_2$ restricts H_1 and H_2 to be disjoint while giving a name H to the conjoined heaps $H_1 * H_2$.

As in [13], $\mathcal{H}$ is then extended with *user-defined* recursive predicates. We use the framework of *Constraint Logic Programming* (CLP) [16] to inherit its syntax, semantics, and most importantly, its built-in notions of unfolding rules. For brevity, we just informally explain the language. The following rules constitutes a recursive definition of predicate $\mathsf{list}(x, L)$, which specifies a skeleton *list*.

$$\begin{array}{lll} \mathsf{list}(x,L) & \text{:-} & x = 0,\ L \simeq \Omega. \\ \mathsf{list}(x,L) & \text{:-} & L \simeq (x{\mapsto}t){*}L_1,\ \mathsf{list}(t, L_1). \end{array}$$

The *semantics* of a set of rules is traditionally known as the "least model" semantics (LMS). Essentially, this is the set of groundings of the predicates which are true when the rules are read as traditional implications. The rules above dictates that all true groundings of $\mathsf{list}(x, L)$ are such that x is an integer, L is a heap which contains a skeleton list starting from x. More specifically, when the list is empty, the root node is equal to `null` ($x = 0$), and the heap is empty ($L \simeq \Omega$). Otherwise, we can split the heap L into two disjoint parts: a singleton heap $(x{\mapsto}t)$ and the remaining heap L_1, where L_1 corresponds to the heap that contains a skeleton list starting from t.

We now provide the definitions for list segments, which will be used in our later examples. Do note the extra explicit heap variable L, in comparison with corresponding definitions in SL.

$$\begin{array}{lll} \widehat{\mathsf{ls}}(x,y,L) & \text{:-} & x{=}y,\ L \simeq \Omega. \\ \widehat{\mathsf{ls}}(x,y,L) & \text{:-} & x{\neq}y,\ L \simeq (x{\mapsto}t){*}L_1,\ \widehat{\mathsf{ls}}(t,y,L_1). \\ & & \\ \mathsf{ls}(x,y,L) & \text{:-} & x{=}y,\ L \simeq \Omega. \\ \mathsf{ls}(x,y,L) & \text{:-} & x{\neq}y,\ L \simeq (t{\mapsto}y){*}L_1,\ \mathsf{ls}(x,t,L_1). \end{array}$$

We also emphasize that the main advantage of this language is the possibility of deriving the strongest postcondition along each program path. It is indeed the main contribution of [13]. Specifically, in order to prove the Hoare triple $\{\phi\}S\{\psi\}$ for a loop-free program S, we simply generate strongest postcondition ψ' along each of its straight-line paths and obtain the verification condition $\psi' \models \psi$. Note that the handling of loops can be reduced to this loop-free setting because of user-specified invariants. For procedure calls, we still make use of the (standard) frame rule to generate proof obligations. We put forward that, in all our experiments (Section 5), the verification conditions are generated using the frame rule (manually though) and the symbolic execution rules of [13].

4. The Proof Method

Background on CLP: This is provided for the convenience of the readers. An *atom* is of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and $\tilde{t}$ is a tuple of $\mathcal{H}$ terms. A *rule* is of the form $A\,\text{:-}\,\Psi, \tilde{B}$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint Ψ constitute the *body* of the rule. A finite set of rules is then used to define a predicate. A *goal* has exactly the same format as the body of a rule. A goal that contains only constraints and no atoms is called *final*.

A *substitution* θ simultaneously replaces each variable in a term or constraint e into some expression, and we write $e\theta$ to denote the result. A *renaming* is a substitution which maps each variable in the expression into a distinct variable. A *grounding* is a substitution which maps each variable into its intended universe of discourse: an integer or a heap, in the case of our CLP($\mathcal{H}$). Where Ψ is a constraint, a grounding of Ψ results in *true* or *false* in the usual way.

A *grounding* θ of an atom $p(\tilde{t})$ is an object of the form $p(\tilde{t}\theta)$ having no variables. A grounding of a goal $\mathcal{G} \equiv (p(\tilde{t}), \Psi)$ is a grounding θ of $p(\tilde{t})$ where $\Psi\theta$ is *true*. We write $[\![\mathcal{G}]\!]$ to denote the set of groundings of $\mathcal{G}$.

Let $\mathcal{G} \equiv (B_1, \cdots, B_n, \Psi)$ and P denote a non-final goal and a set of rules respectively. Let $R \equiv A\,\text{:-}\,\Psi_1, C_1, \cdots, C_m$ denote a rule in P, written so that none of its variables appear in $\mathcal{G}$. Let the equation $A = B$ be shorthand for the pairwise equation of the corresponding arguments of A and B. A *reduct* of $\mathcal{G}$ using a clause R, denoted $reduct(\mathcal{G}, R)$, is of the form
$(B_1, \cdots, B_{i-1}, C_1, \cdots, C_m, B_{i+1}, \cdots, B_n, B_i = A, \Psi, \Psi_1)$
provided the constraint $B_i = A \wedge \Psi \wedge \Psi_1$ is satisfiable.

A *derivation sequence* for a goal $\mathcal{G}_0$ is a possibly infinite sequence of goals $\mathcal{G}_0, \mathcal{G}_1, \cdots$, where $\mathcal{G}_i,\ i > 0$ is a reduct of $\mathcal{G}_{i-1}$. A *derivation tree* for a goal is defined in the obvious way.

DEFINITION 1 (Unfold). *Given a program P and a goal $\mathcal{G}$:* $\textsc{unfold}(\mathcal{G})$ *is* $\{\mathcal{G}' | \exists R \in P : \mathcal{G}' = reduct(\mathcal{G}, R)\}$. $\square$

Given a goal $\mathcal{L}$ and an atom $p \in \mathcal{L}$, $\textsc{unfold}_p(\mathcal{L})$ denotes the set of formulas transformed from $\mathcal{L}$ by unfolding p.

DEFINITION 2 (Entailment). *An* entailment *is of the form* $\mathcal{L} \models \mathcal{R}$, *where* $\mathcal{L}$ *and* $\mathcal{R}$ *are goals.* $\square$

This paper considers proving the validity of the entailment $\mathcal{L} \models \mathcal{R}$ under a given program P. This entailment means that $lm(P) \models (\mathcal{L} \rightarrow \mathcal{R})$, where $lm(P)$ denotes the "least model" of the program P which defines the recursive predicates — called *assertion* predicates — occurring in $\mathcal{L}$ and $\mathcal{R}$. This is simply the set of all groundings of atoms of the assertion predicates which are *true* in P. The expression $(\mathcal{L} \rightarrow \mathcal{R})$ means that, for each grounding θ of $\mathcal{L}$ and $\mathcal{R}$, $\mathcal{L}\theta$ is in $lm(P)$ implies that so is $\mathcal{R}\theta$.

4.1 Unfold and Match (U+M)

Assume that we start off with $\mathcal{L} \models \mathcal{R}$. If this entailment can be proved directly, by unification and/or consulting an off-the-shelf SMT solver, we say that the entailment is trivial: a *direct proof* is obtained even without considering the "meaning" of the recursively defined predicates (they are treated as *uninterpreted*). When it is not the case — the entailment is non-trivial — a standard approach is to apply unfolding/folding until all the "frontier" become trivial. We note that, in our framework, we perform only unfolding, but now to both the LHS (the antecedent) and the RHS (the consequent) of the entailment. The effect of unfolding the RHS is similar to a folding operation on the LHS. In more detail, when direct proof fails, U+M paradigm proceeds in two possible ways:

- First, select a recursive atom $p \in \mathcal{L}$, unfold $\mathcal{L}$ wrt. p and obtain the goals $\mathcal{L}_1, \ldots, \mathcal{L}_n$. The validity of the original entailment can now be obtained by ensuring the validity of *all* the entailments $\mathcal{L}_i \models \mathcal{R}$ $(1 \leq i \leq n)$.
- Second, select a recursive atom $q \in \mathcal{R}$, unfold $\mathcal{R}$ wrt. q and obtain the goals $\mathcal{R}_1, \ldots, \mathcal{R}_m$. The validity of the original entailment can now be obtained by ensuring the validity of *any one of* the entailments $\mathcal{L} \models \mathcal{R}_j$ $(1 \leq j \leq m)$.

So the proof process can proceed recursively either by proving *all* $\mathcal{L}_i \models \mathcal{R}$ or by proving *one* $\mathcal{L} \models \mathcal{R}_j$ for some j. Since the original LHS and RHS usually contain more than one recursive atoms, this proof process naturally triggers a search tree. Termination can be guaranteed by simply bounding the maximum number of left and right unfolds allowed. In practice, the number of recursive atoms used in an entailment is usually small, thus resulting tree size is often manageable.

4.2 Formula Re-writing with Dynamic Induction Hypotheses

We now present a formal calculus for the proof of $\mathcal{L} \models \mathcal{R}$ that goes beyond unfold-and-match. The power of our proof framework comes from the key concept: *induction*.

DEFINITION 3 (Proof Obligation). *A* proof obligation *is of the form* $\tilde{A} \vdash \mathcal{L} \models \mathcal{R}$ *where* $\mathcal{L}$ *and* $\mathcal{R}$ *are goals and* $\tilde{A}$ *is a set of pairs* $\langle A; p \rangle$, *where* A *is an* assumed *entailment and* p *is a* recursive atom. $\square$

The role of proof obligations is to capture the state of the proof process. Each element in $\tilde{A}$ is a pair, of which the first is an entailment A whose truth can be assumed inductively. A acts as an (dynamically generated) induction hypothesis and can be used to transform subsequently encountered obligations in the proof path. The second is a recursive atom p, to which the application of a left unfold gives rise to the addition of the induction hypothesis A.

Our proof rules – the obligation at the bottom, and its reduced form on top – are presented in Fig. 6. Given $\mathcal{L} \models \mathcal{R}$, our proof shall start with $\emptyset \vdash \mathcal{L} \models \mathcal{R}$, and proceed by repeatedly applying these rules. Each rule operates on a proof obligation. In this process, the proof obligation may be discharged (indicated by `True`); or new proof obligation(s) may be produced. $\mathcal{L} \models_{\texttt{SMT}} \mathcal{R}$ denotes the validity of $\mathcal{L} \models \mathcal{R}$ is obtained by consulting a generic SMT solver.

• The *substitution* (SUB) rule removes one occurrence of an assertion predicate, say atom $p(\tilde{y})$, appearing in the RHS of a proof obligation. Applying the (SUB) rule repeatedly will ultimately reduce a proof obligation to the form which contains no recursive atoms in the RHS, while at the same time (hopefully) most existential variables on the RHS are eliminated. Then, the *constraint proof* (CP) rule may be attempted by simply treating all remaining recursive atoms (in the LHS) as uninterpreted and by applying the underlying theory solver assumed in the language we use.

The combination of (SUB) and (CP) rules attempts, what we call, a *direct proof.* In principle, it is similar to the process of "matching" in the U+M paradigm. For brevity we then use $\mathcal{L} \models_{\texttt{DP}} \mathcal{R}$ to denote the fact that the validity of $\mathcal{L} \models \mathcal{R}$ can be proved directly using only (SUB) and (CP) rules.

• The *left unfold with induction hypothesis* (LU+I) is a key rule. It selects a recursive atom p on the LHS and performs a complete unfold of the LHS wrt. the atom p, producing a new set of proof obligations. The original obligation, while being removed, is added as an assumption to every newly produced proof obligation, opening the door for the later being used as an induction hypothesis. For technical reason needed below, we do not just add the obligation $\mathcal{L} \models \mathcal{R}$ as an assumption, but also need to keep track of the atom p. This is why in the rule we see a pair $\langle \mathcal{L} \models \mathcal{R}; p \rangle$ added into the current set of assumptions $\tilde{A}$.

On the other hand, the *right unfold* (RU) rule selects some recursive atom q and performs an unfold on the RHS of a proof obligation wrt. q. In the proof process, the two unfold rules will be systematically interleaved.

EXAMPLE 1. *Consider the following proof obligation:*

$$\tilde{A} \vdash \mathsf{list}(x, L) \models \mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L \bumpeq L_1{*}L_2\,.$$

$$(\textsc{sub})\;\dfrac{(\textsc{ru})\;\dfrac{(\textsc{cp})\;\dfrac{\texttt{True}}{\tilde{A} \vdash \mathsf{list}(x, L) \models x = x, L_1 \bumpeq \Omega, L \bumpeq L_1{*}L}}{\tilde{A} \vdash \mathsf{list}(x, L) \models \mathsf{ls}(x, x, L_1), L \bumpeq L_1{*}L}}{\tilde{A} \vdash \mathsf{list}(x, L) \models \mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L \bumpeq L_1{*}L_2}$$

Figure 7: Proving with just U+M

In Fig. 7, we show how this proof obligation can be successfully dispensed by applying (SUB), (RU), and (CP) rules in sequence. Note how the (SUB) rule binds the existential variable y to x, simplifying the RHS of the proof obligation.

• The *induction applications*, namely (IA-1) and (IA-2) rules, transform the current obligation by making use of an assumption which has been added by the (LU+I) rule. The two rules, also called the "induction rules" for short, allow us to treat previously encountered obligations as possible induction hypotheses.

Instead of directly proving the current obligation $\mathcal{L} \models \mathcal{R}$, we now proceed by finding $\overline{\mathcal{L}}$ and $\overline{\mathcal{R}}$ such that $\mathcal{L} \models \overline{\mathcal{L}} \models \overline{\mathcal{R}} \models \mathcal{R}$. The key here is to find those candidate goals where the validity of $\overline{\mathcal{L}} \models \overline{\mathcal{R}}$ directly follows from a "similar" assumption A, together with θ to rename all the variables in A to the variables in the current obligation, namely $\mathcal{L} \models \mathcal{R}$. Assumption A is an obligation which has been previously encountered in the proof process, and $A\theta$ assumed to be true, as an induction hypothesis. Particularly, we choose $\overline{\mathcal{L}}$ and $\overline{\mathcal{R}}$ so we can (easily) find a renaming θ such that $A\theta \implies \overline{\mathcal{L}} \models \overline{\mathcal{R}}$ ($\implies$ denotes logical implication).

$$(\textsc{cp})\quad \frac{\texttt{True}}{\tilde{A} \vdash \mathcal{L} \models \mathcal{R}} \quad \mathcal{L} \models_{\texttt{SMT}} \mathcal{R}\text{, where recursive atoms are treated as uninterpreted}$$

$$(\textsc{sub})\quad \frac{\tilde{A} \vdash \mathcal{L} \wedge p(\tilde{x}) \models \mathcal{R}\theta}{\tilde{A} \vdash \mathcal{L} \wedge p(\tilde{x}) \models \mathcal{R} \wedge p(\tilde{y})} \quad \begin{array}{l}\text{there exists a substitution } \theta \text{ for} \\ \text{existential variables in } \tilde{y} \text{ s.t. } \mathcal{L} \wedge p(\tilde{x}) \models_{\texttt{SMT}} \tilde{x} = \tilde{y}\theta\end{array}$$

$$(\textsc{lu+i})\quad \frac{\bigcup_{i=1}^{n}\{\tilde{A} \cup \{\langle \mathcal{L} \models \mathcal{R}; p\rangle\} \vdash \mathcal{L}_i \models \mathcal{R}\}}{\tilde{A} \vdash \mathcal{L} \models \mathcal{R}} \quad \begin{array}{l}\text{Select an atom } p \in \mathcal{L} \text{ and} \\ \textsc{unfold}_p(\mathcal{L}) = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}\end{array}$$

$$(\textsc{ru})\quad \frac{\tilde{A} \vdash \mathcal{L} \models \mathcal{R}'}{\tilde{A} \vdash \mathcal{L} \models \mathcal{R}} \quad \begin{array}{l}\text{Select an atom } q \in \mathcal{R} \text{ and} \\ \mathcal{R}' \in \textsc{unfold}_q(\mathcal{R})\end{array}$$

$$(\textsc{ia-1})\quad \frac{\tilde{A} \vdash \mathcal{R}'\theta \wedge \mathcal{L}_2 \models \mathcal{R}}{\tilde{A} \vdash p(\tilde{x}) \wedge \mathcal{L}_1 \wedge \mathcal{L}_2 \models \mathcal{R}} \quad \begin{array}{l}\langle p(\tilde{y}) \wedge \mathcal{L}' \models \mathcal{R}'; p(\tilde{y})\rangle \in \tilde{A} \text{ and } \texttt{gen}(p(\tilde{x})) \geq \texttt{kill}(p(\tilde{y})), \\ \text{there exists a renaming } \theta \text{ s.t. } \tilde{x} = \tilde{y}\theta \text{ and } \mathcal{L}_1 \models_{\texttt{DP}} \mathcal{L}'\theta\end{array}$$

$$(\textsc{ia-2})\quad \frac{\tilde{A} \vdash \mathcal{L}_1 \models \mathcal{L}'\theta}{\tilde{A} \vdash p(\tilde{x}) \wedge \mathcal{L}_1 \models \mathcal{R}} \quad \begin{array}{l}\langle p(\tilde{y}) \wedge \mathcal{L}' \models \mathcal{R}'; p(\tilde{y})\rangle \in \tilde{A} \text{ and } \texttt{gen}(p(\tilde{x})) \geq \texttt{kill}(p(\tilde{y})) \\ \text{and there exists a renaming } \theta \text{ s.t. } \tilde{x} = \tilde{y}\theta \text{ and } \mathcal{R}'\theta \models_{\texttt{DP}} \mathcal{R}\end{array}$$

Figure 6: General Proof Rules

To be more deterministic and to prevent us from transforming to obligations harder than the original one, we require that at least one of the remaining two entailments, namely $\mathcal{L} \models \overline{\mathcal{L}}$ and $\overline{\mathcal{R}} \models \mathcal{R}$, is discharged quickly by a direct proof.

In (IA-1) rule, given the current obligation $p(\tilde{x}) \wedge \mathcal{L}_1 \wedge \mathcal{L}_2 \models \mathcal{R}$ and an assumption $A \equiv p(\tilde{y}) \wedge \mathcal{L}' \models \mathcal{R}'$, we choose $p(\tilde{x}) \wedge \mathcal{L}'\theta \wedge \mathcal{L}_2$ to be our $\overline{\mathcal{L}}$ and $\mathcal{R}'\theta \wedge \mathcal{L}_2$ to be our $\overline{\mathcal{R}}$. We can see that the validity of $\overline{\mathcal{L}} \models \overline{\mathcal{R}}$ directly follows from the assumption $A\theta$. One restriction onto the renaming θ, to avoid circular reasoning, is that θ must rename $\tilde{y}$ to $\tilde{x}$ where $p(\tilde{x})$ is an atom which has been generated after $p(\tilde{y})$ had been unfolded. Such fact is indicated by $\texttt{gen}(p(\tilde{x})) \geq \texttt{kill}(p(\tilde{y}))$ in our rule. While $\texttt{gen}(p)$ denotes the timestamp when the recursive atom p is generated during the proof process, $\texttt{kill}(p)$ denotes the timestamp when p is unfolded and removed. Another side condition for this rule is that the validity of $\mathcal{L} \models \overline{\mathcal{L}}$, or equivalently, $\mathcal{L}_1 \models \mathcal{L}'\theta$ is discharged immediately by a direct proof.

In (IA-2) rule, given the current obligation $p(\tilde{x}) \wedge \mathcal{L}_1 \models \mathcal{R}$ and an assumption $A \equiv p(\tilde{y}) \wedge \mathcal{L}' \models \mathcal{R}'$, on the other hand, $p(\tilde{y})\theta \wedge \mathcal{L}'\theta$ serves as our $\overline{\mathcal{L}}$ while $\mathcal{R}'\theta$ serves as our $\overline{\mathcal{R}}$. The validity of $\overline{\mathcal{L}} \models \overline{\mathcal{R}}$ trivially follows from the assumption $A\theta$, namely $p(\tilde{x}) \wedge \mathcal{L}'\theta \models \mathcal{R}'\theta$. As in (IA-1), we also put similar restriction upon the renaming θ. Another side condition we require is that the validity of $\overline{\mathcal{R}} \models \mathcal{R}$ can be discharged immediately by a direct proof. At this point we could see the duality nature of (IA-1) and (IA-2).

Now let us briefly and intuitively explain the restriction upon the renaming θ. Here we make sure that θ renames atom $p(\tilde{y})$ to atom $p(\tilde{x})$, where $p(\tilde{x})$ has been generated after $p(\tilde{y})$ had been unfolded (and removed). This helps to rule out certain potential θ which does not correspond to a number of left unfolds. Such restriction helps ensure *progressiveness* in the proof process before the induction rules can take place. Otherwise, assuming the truth of $A\theta$ in constructing the proof for A might not be valid. This is the reason why for each element of $\tilde{A}$, we not only keep track of the assumption, but also the recursive atom p to which the application of (LU+I) gives rise to the addition of such assumption.

It is important to note that, our framework as it stands, does not require any consideration of a base case, nor any well-founded measure. Instead, we depend on the Least Model Semantics (LMS) of our assertion language and the above-mentioned restrictions on the renaming θ. In other words, by constraining the use of the rules, which is transparent to the user, we guarantee to achieve a well-founded conclusion.

Least Model Semantics: Let us now give an example to illustrate why our proof is working under the LMS. Consider the recursive predicate $\mathtt{p}$, defined as

```
p(x) :-  p(x).
```

and the following two proof obligations:

$$\mathsf{p}(x) \models \mathsf{list}(x, L) \tag{4.1}$$

$$\mathsf{list}(x, L) \models \mathsf{p}(x) \tag{4.2}$$

We will now demonstrate that our method can prove (4.1), but not (4.2). We remark that (4.1) holds because under the LMS, the LHS has no model; therefore no refutation can be found regardless of what the RHS is. In other words, *false* implies anything. On the other hand, (4.2) does not hold because $x = 0$ (and $L \mathrel{\hat{-}} \Omega$) is a model of the LHS, but not a model of the RHS.

$$\begin{array}{rc} & \texttt{True} \\ \cline{2-2} (\textsc{cp}) & \{A\} \vdash \mathsf{list}(x,L) \models \mathsf{list}(x,L) \\ \cline{2-2} (\textsc{ia-1}) & \{A\} \vdash \mathsf{p}(x) \models \mathsf{list}(x,L) \\ \cline{2-2} (\textsc{lu+i}) & \emptyset \vdash \mathsf{p}(x) \models \mathsf{list}(x,L) \end{array}$$

Figure 8: Our Proof for (4.1)

Fig. 8 shows how our method would handle (4.1). We first perform a left unfolding, adding $A \equiv \langle \mathsf{p}(x) \models \mathsf{list}(x, L); \mathsf{p}(x)\rangle$ into the set of assumptions. Note that this unfolding step kills the predicate $\mathrm{p}(x)$ and generates a new predicate $\mathrm{p}(x)$. Thus the rule (IA-1) is applicable now. We then re-write the LHS from $\mathrm{p}(x)$ to $\mathsf{list}(x, L)$. Finally the proof succeeds by consulting constraint solver, treating $\mathsf{list}(x, L)$ as uninterpreted.

In contrast, now consider obligation (4.2) in Fig. 9.

$$\begin{array}{rcc} & \{A'\} \vdash x = 0, L \mathrel{\hat{-}} \Omega \models \mathsf{p}(x) & \vdots \\ \cline{2-3} (\textsc{lu+i}) & \multicolumn{2}{c}{\emptyset \vdash \mathsf{list}(x,L) \models \mathsf{p}(x)} \\ \cline{2-3} (\textsc{ru}) & \multicolumn{2}{c}{\emptyset \vdash \mathsf{list}(x,L) \models \mathsf{p}(x)} \end{array}$$

Figure 9: An Unsuccessful Attempt for (4.2)

Obviously, a direct proof for this is not successful. However, if we proceed by a right unfold first, we get back the same obligation. Different from before, and importantly, now no new assumption is added. We can see that the step does not help us progress and therefore performing right unfold repetitively would get us nowhere. Now consider performing a left unfold on the obligation. The proof succeeds if we can discharge both

$$\{A'\} \vdash x = 0, L \doteq \Omega \models p(x) \texttt{ and}$$
$$\{A'\} \vdash L \doteq (x{\mapsto}t){*}L_1, \mathsf{list}(t, L_1) \models p(x),$$

where $A' \equiv \langle \mathsf{list}(x, L) \models \mathsf{p}(x); \mathsf{list}(x, L)\rangle$.

Focus on the obligation $\{A'\} \vdash x = 0, L \doteq \Omega \models p(x)$. Clearly consulting a constraint solver or performing substitution does not help. Rule (LU+I) is not applicable since no recursive predicate on the LHS. As before, we cannot progress using (RU) rule. Importantly, the side conditions prevent (IA-1) and (IA-2) from taking place. In summary, with our proof rules, this (wrong fact) cannot be established.

4.3 Proving the Two Motivating Examples

Let us now revisit the two motivating examples introduced earlier, on which both U+M and "Cyclic Proof" are not effective. The main reason is that both examples involve unmatchable predicates while at the same time exhibiting "recursion divergence".

EXAMPLE 2. *Consider the entailment relating two definitions of list segments:* $\widehat{\mathsf{ls}}(x, y, L) \models \mathsf{ls}(x, y, L)$.

Our method can discharge this obligation by applying (IA-1) rule twice. For space reason, in Fig. 10, we only show the interesting path of the proof tree (leftmost position). First, we unfold the predicate $\widehat{\mathsf{ls}}(x, y, L)$ in the LHS of the given obligation via (LU+I) rule. The original obligation, while being removed, is added as an assumption A_1. We next make use of A_1 as an induction hypothesis to perform a re-writing step, i.e., an application of (IA-1) rule. Similarly, in the third step, we unfold the predicate $\mathsf{ls}(t, y, L_1)$ in the LHS via (LU+I) rule and add the assumption A_2. After unfolding in the RHS via (RU) rule and re-writing with the induction hypothesis A_2 using (IA-1) rule, we are able to bind the existential variable z_1 to z and simplify both sides of the proof obligation using (SUB) rule. Finally, the proof path is terminated by consulting a constraint solver, i.e., using (CP) rule.

EXAMPLE 3. *Consider the entailment:*
$\mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L_1{*}L_2 \models \mathsf{list}(x, L), L \doteq L_1{*}L_2.$

Fig. 11 shows, only the interesting proof path, how we can successfully prove this entailment using the (IA-2) rule. We first unfold $\mathsf{ls}(x, y, L_1)$ in the LHS, adding A into the set of assumptions. Then using A as an induction hypothesis, we can rewrite the current obligation via (IA-2) rule. Note that, here we use (IA-2) rule instead of (IA-1) rule as in previous example. After applying (RU) rule, we are able to bind the existential variable y_1 to y and simplify both sides of the proof obligation with (SUB) rule. Finally, the proof path is terminated by consulting a constraint solver, i.e., using (CP) rule.

Let us pay a closer attention at the step where we attempt re-writing, making using the available induction hypothesis. For the sake of discussion, assume that instead of (IA-2) we now attempt to apply rule (IA-1). The requirement for θ forces it to rename x to x and y to t. However, the side condition $\mathcal{L}_1 \models_{\mathsf{DP}} \mathcal{L}'\theta$ cannot be fulfilled, since

$$x \neq y, L_1 \doteq (t{\mapsto}y){*}L_3, \mathsf{list}(y, L_2), L_1{*}L_2 \not\models_{\mathsf{DP}} \mathsf{list}(t, \_).$$

Now return to the attempt of (IA-2) rule. The RHS of the current obligation matches with the RHS of the *only* induction hypothesis perfectly. This matching requires θ to rename x back to x. On the LHS, we further require θ to rename y to t so that $\mathsf{ls}(x, t) \equiv \mathsf{ls}(x, y)\theta$. Note that $\mathsf{ls}(x, t)$ was indeed generated after $\mathsf{ls}(x, y)$ had been unfolded and removed (i.e., killed). The remaining transformation is more straightforward.

THEOREM 1 (Soundness). *The proof method embodied in the rules of Figure 6 is sound.* □

Proof. Please see the technical report [10], where we also present a detailed algorithm.

5. Experiments

In our modular verification framework (with the frame rule), the problem of verifying big programs reduces to proving the kinds of verification conditions addressed in this paper. Our experiments are thus focused on the complexity of the program properties to be proven instead of the size of programs.

Our evaluations are performed on a 3.2GHz Intel processor with 2GB RAM, running Linux. We evaluated our prototype[3] on a comprehensive set of benchmarks, including both academic algorithms and real programs. The benchmarks are collected from existing systems [8, 9, 22, 25, 28], those considered as the state-of-the-art for the purpose of proving user-defined recursive data-structure properties in imperative languages. Some of them are also used in the competition SMT-COMP 2014 (Separation Logic)[4]. Note that, in this competition (where lemmas are discouraged), the benchmarks are of the same scale as ours, though ours contain more benchmarks having shape and data properties intertwined, making previous techniques fail to prove. We first demonstrate our evaluation with benchmarks that the state-of-the-art can handle, then with ones that are beyond their current supports.

5.1 Within the State-of-the-art

In this subsection, we consider the set of proof obligations where the state-of-the-art, e.g., U+M and "Cyclic Proof", are effective. The purpose of this study is to evaluate the *efficiency* of our implementation against existing systems. This exercise serves as a sanity check for our implementation.

We first start with proof obligations where U+M can automatically discharge without the help of user-defined lemmas. They are collected from the benchmarks of U+M frameworks [9, 22, 28]. As expected, our prototype proves all of those obligations; the running time for each is negligible ($\sim$ 0.2 second). This is because the proof obligations usually require just either *one* left unfold or *one* right unfold before matching (a direct proof) can successfully take place.

The second set of benchmarks are from "Cyclic Proof" [8], which are also used in SMT-COMP 2014 (Separation Logic). They are proof obligations which involve unmatchable predicates, thus U+M will not be effective. We also succeed in proving all of those obligations, less than a second for each.

In summary, the results demonstrate that (1) our prototype is able to handle what the state-of-the-art can; (2) our implementation is *competitive* enough.

5.2 Beyond the State-of-the-art

We now move to demonstrate the *key* result of this paper: proving what are beyond the state-of-the-art.

Proving User-Defined Lemmas: Our prototype can prove all commonly used lemmas, collected from [9, 22, 25, 28], which U+M and "Cyclic Proof" cannot handle. The running time is always less than a second for each lemma. Table 1 shows a non-exhaustive list of

[3] The detailed algorithms and our prototype implementation can also be found in our technical report [10].

[4] See https://github.com/mihasighi/smtcomp14-sl

$$\begin{array}{rcl}
 & \texttt{True} & \\ \hline
\text{(CP)} & \{A_1,A_2\} \vdash x{\neq}y, L \bumpeq (x{\mapsto}t){*}L_1, t{\neq}y, L_1 \bumpeq (z{\mapsto}y){*}L_2 \models x{\neq}y, L \bumpeq (z{\mapsto}y){*}(x{\mapsto}t){*}L_2 & \\ \hline
\text{(SUB)} & \{A_1,A_2\} \vdash x{\neq}y, L \bumpeq (x{\mapsto}t){*}L_1, t{\neq}y, L_1 \bumpeq (z{\mapsto}y){*}L_2, \mathsf{ls}(x,z,(x{\mapsto}t){*}L_2) \models x{\neq}y, L \bumpeq (z_1{\mapsto}y){*}L_3, \mathsf{ls}(x,z_1,L_3) & \\ \hline
\text{(IA-1)} & \{A_1,A_2\} \vdash x{\neq}y, L \bumpeq (x{\mapsto}t){*}L_1, t{\neq}y, L_1 \bumpeq (z{\mapsto}y){*}L_2, \mathsf{ls}(t,z,L_2) \models x{\neq}y, L \bumpeq (z_1{\mapsto}y){*}L_3, \mathsf{ls}(x,z_1,L_3) & \\ \hline
\text{(RU)} & \{A_1,A_2\} \vdash x{\neq}y, L \bumpeq (x{\mapsto}t){*}L_1, t{\neq}y, L_1 \bumpeq (z{\mapsto}y){*}L_2, \mathsf{ls}(t,z,L_2) \models \mathsf{ls}(x,y,L) & \vdots \\ \hline
\text{(LU+I)} & \{A_1\} \vdash x{\neq}y, L \bumpeq (x{\mapsto}t){*}L_1, \mathsf{ls}(t,y,L_1) \models \mathsf{ls}(x,y,L) & \\ \hline
\text{(IA-1)} & \{A_1\} \vdash x{\neq}y, L \bumpeq (x{\mapsto}t){*}L_1, \widehat{\mathsf{ls}}(t,y,L_1) \models \mathsf{ls}(x,y,L) & \vdots \\ \hline
\text{(LU+I)} & \emptyset \vdash \widehat{\mathsf{ls}}(x,y,L) \models \mathsf{ls}(x,y,L) &
\end{array}$$

where $A_1 \equiv \langle \widehat{\mathsf{ls}}(x,y,L) \models \mathsf{ls}(x,y,L); \widehat{\mathsf{ls}}(x,y,L)\rangle$ **and** $A_2 \equiv \langle x{\neq}y, L \bumpeq (x{\mapsto}t){*}L_1, \mathsf{ls}(t,y,L_1) \models \mathsf{ls}(x,y,L); \mathsf{ls}(t,y,L_1)\rangle$

Figure 10: Proving $\widehat{\mathsf{ls}}(x,y,L) \models \mathsf{ls}(x,y,L)$.

$$\begin{array}{rcl}
 & \texttt{True} & \\ \hline
\text{(CP)} & \{A\} \vdash x \neq y, L_1 \bumpeq (t{\mapsto}y){*}L_3, \mathsf{list}(y,L_2), L_1{*}L_2 \models L_4 \bumpeq (t{\mapsto}y){*}L_2, L_1{*}L_2 \bumpeq L_3{*}L_4 & \\ \hline
\text{(SUB)} & \{A\} \vdash x \neq y, L_1 \bumpeq (t{\mapsto}y){*}L_3, \mathsf{list}(y,L_2), L_1{*}L_2 \models L_4 \bumpeq (t{\mapsto}y_1){*}L_5, \mathsf{list}(y_1,L_5), L_1{*}L_2 \bumpeq L_3{*}L_4 & \\ \hline
\text{(RU)} & \{A\} \vdash x \neq y, L_1 \bumpeq (t{\mapsto}y){*}L_3, \mathsf{list}(y,L_2), L_1{*}L_2 \models \mathsf{list}(t,L_4), L_1{*}L_2 \bumpeq L_3{*}L_4 & \\ \hline
\text{(IA-2)} & \{A\} \vdash x \neq y, L_1 \bumpeq (t{\mapsto}y){*}L_3, \mathsf{ls}(x,t,L_3), \mathsf{list}(y,L_2), L_1{*}L_2 \models \mathsf{list}(x,L), L \bumpeq L_1{*}L_2 & \vdots \\ \hline
\text{(LU+I)} & \emptyset \vdash \mathsf{ls}(x,y,L_1), \mathsf{list}(y,L_2), L_1{*}L_2 \models \mathsf{list}(x,L), L \bumpeq L_1{*}L_2 &
\end{array}$$

where $A \equiv \langle \mathsf{ls}(x,y,L_1), \mathsf{list}(y,L_2), L_1{*}L_2 \models \mathsf{list}(x,L), L \bumpeq L_1{*}L_2; \mathsf{ls}(x,y,L_1)\rangle$

Figure 11: Proving $\mathsf{ls}(x,y,L_1), \mathsf{list}(y,L_2), L_1{*}L_2 \models \mathsf{list}(x,L), L \bumpeq L_1{*}L_2$.

common user-defined lemmas. We purposely abstract them from the original usage in order to make them general and representative enough. The lemmas are written in traditional Separation Logic syntax for succinctness. Note that due to the duality of the definitions for list segments, e.g., ls vs. $\widehat{\mathsf{ls}}$, each lemma containing them would usually has a dual version, which for space reason we do not list down in Table 1. Similarly, some extensions, e.g., to capture the relationship of collective data values (using sets or sequences) between the LHS and the RHS, while can be automatically discharged by our prototype, are not listed in the table.

Table 1. Proving Lemmas (existing systems cannot prove).

Lemma
$\mathsf{sorted\_list}(x, min) \models \mathsf{list}(x)$
$\mathsf{sorted\_list_1}(x, len, min) \models \mathsf{list_1}(x, len)$
$\mathsf{sorted\_list_1}(x, len, min) \models \mathsf{sorted\_list}(x, min)$
$\mathsf{sorted\_ls}(x, y, min, max) * \mathsf{sorted\_list}(y, min_2) \wedge max \leq min_2 \models \mathsf{sorted\_list}(x, min)$
$\mathsf{ls}(x,y) * \mathsf{list}(y) \models \mathsf{list}(x)$
$\mathsf{ls}(x,y) \models \widehat{\mathsf{ls}}(x,y)$ and $\widehat{\mathsf{ls}}(x,y) \models \mathsf{ls}(x,y)$
$\widehat{\mathsf{ls}}_1(x,y,len_1) * \widehat{\mathsf{ls}}_1(y,z,len_2) \models \widehat{\mathsf{ls}}_1(x,z,len_1{+}len_2)$
$\mathsf{ls_1}(x,y,len_1) * \mathsf{list_1}(y,len_2) \models \mathsf{list_1}(x,len_1{+}len_2)$
$\widehat{\mathsf{ls}}_1(x,last,len) * (last \mapsto new) \models \widehat{\mathsf{ls}}_1(x,new,len+1)$
$\mathsf{dls}(x,y) * \mathsf{dlist}(y) \models \mathsf{dlist}(x)$
$\widehat{\mathsf{dls}}_1(x,y,len_1) * \widehat{\mathsf{dls}}_1(y,z,len_2) \models \widehat{\mathsf{dls}}_1(x,z,len_1{+}len_2)$
$\mathsf{dls_1}(x,y,len_1) * \mathsf{dlist_1}(y,len_2) \models \mathsf{dlist_1}(x,len_1{+}len_2)$
$\mathsf{avl}(x,hgt,min,max,balance) \models \mathsf{bstree}(x,hgt,min,max)$
$\mathsf{bstree}(x,height,min,max) \models \mathsf{bintree}(x,height)$

Let us briefly comment on Table 1. The first group talks about sorted linked lists. As an example, the second lemma is to state that a sorted list with length len and the minimum element min is also a list with the same length. The second, third and fourth groups are related to singly-linked lists, doubly-linked lists, and trees respectively.

Verifying Programs without Using Lemmas: Lemmas can serve many purposes. One of its important usage in U+M systems is to equip a proof system with the power of user-provided re-writing rules, to overcome the main limitation of unfold-and-match. However, in the context of program verification, eliminating the usage of lemmas is crucial for improving the performance, because lemma applications, coupled with unfolding, often induce large search space.

We now use a subset of academic algorithms and open-source library programs[5], collected and published by [9, 28], to demonstrate that our prototype can verify these programs without even stating the appropriate lemmas. The library programs include Glib open source library, the OpenBSD library, the Linux kernel, the memory regions and the page cache implementations from two different operating systems. While Table 2 summarizes the verification of data structures from academic algorithms, Table 3 reports on open-source library programs.

Table 2. Verification of Academic Algorithms (existing systems require lemmas).

DS	Function	T/F
`Sorted List`	`find_last_iter, insert_iter, quick_sort_iter, bubble_sort`	$<1s$
`Circular List`	`count`	$<1s$
`BST`	`insert_iter,find_leftmost_iter remove_root_iter, delete_iter`	$<1s$

Remark #1: Using automatic induction, we have successfully eliminated the requirement for lemmas in existing systems (e.g., [9, 28]) for proving the functional correctness of the programs in Table 2 and 3. As already stated in Section 1, existing systems require lemmas in two common scenarios. First, it is when the traversal order of the data structures is different from what suggested by the recursive definitions, e.g., `OpenBSD/queue.h`. Second, it is due to the boundaries caused by iterative loops or multiple function calls. One example is `append` function in `glib/gslist.c`, where (in addition to the list definition) the list segment, ls(head,last), is necessary to say about the function invariant — the last node of a non-empty input list is always reachable from the list's head. Other examples are to make a connection between a sorted list and a singly-linked list (e.g., in sorting algorithms), between two sorted partitions (e.g. in `quick_sort_iter`), between a circular list and a list segment (e.g., `count`), etc.

[5] See `http://www.cs.uiuc.edu/∼madhu/dryad/sl`

Table 3. Verification of Open-Source Libraries (existing systems require lemmas).

Program	Function	T/F
glib/gslist.c `Singly Linked-List`	`find, position, index, nth,last,length,append, insert_at_pos,merge_sort, remove,insert_sorted_list`	$<1s$
glib/glist.c `Doubly Linked-List`	`nth, position, find, index, last, length`	$<1s$
OpenBSD/ queue.h `Queue`	`simpleq_remove_after, simpleq_insert_tail, simpleq_insert_after`	$<1s$
ExpressOS/ cachePage.c	`lookup_prev, add_cachepage`	$<1s$
linux/mmap.c	`insert_vm_struct`	$<1s$

Remark #2: The verification time for each function is always less than 1 second. This is within our expectation because whenever our proof method succeeds, the size of the proof tree is relatively small. For example, in order to prove the functional correctness of `append` function in `glib/gslist.c`, we only need to prove 3 obligations, each of which requires no more than two left unfolds, two right unfolds and two inductions[6]. In fact, the maximum number of left unfolds, right unfolds and inductions used in our system are 5, 5 and 3 respectively, even for the functions that take U+M frameworks much longer time to prove. For example, consider `simpleq_insert_after`, a function to insert an element into a queue. This example requires reasoning about unmatchable predicates: to prove it DRYAD needs 18 seconds and the help from a lemma. Such inefficiency is due to the use of a complicated lemma[7], which consists of a large disjunction. Though efficient in practice, SMT solvers still face a combinatorial explosion challenge as they dissect the disjunction. In other words, in addition to having a higher level of automation, our framework has a potential advantage of being more efficient than existing U+M systems.

6. Related Work

There is a vast literature on program verification considering data structures. The well known formalism of Separation Logic (SL) [33] is often combined with a recursive formulation of data structure properties. Implementations, however, are incomplete, e.g., [2, 14], or deal only with fragments [1, 23]. There is also literature on decision procedures for restricted heap logics; we mention just a few examples: [3, 4, 18, 29–31]. These have, however, severe restrictions on expressivity. None of them can handle the VC's of the kind considered in this paper.

There is also a variety of verification tools based on classical logics and SMT solvers. Some examples are Dafny [20], VCC [11] and Verifast [15] which require significant ghost annotations, and annotations that explicitly express and manipulate frames. They do not automatically verify the general and complex obligations addressed in this paper; but such obligations are often resorted to interactive theorem provers, e.g., Mona, Isabelle or Coq, enabling manual guidance from the users.

Navarro and Rybalchenko showed that significant performance improvements can be obtained by incorporating first-order theorem proving techniques into SL provers [24]. However, the focus of that work is about list segments, not general user-defined recursive predicates. On a similar thread, [27] advances the automation of SL, using SMT, in verifying procedures manipulating list-like data structures. The works [9, 22, 28, 36, 37] are also closely related: they form the U+M paradigm which we have carefully discussed in Section 1 and 2.

In the literature, there have been works on automatic induction [5, 12, 21, 35]. They are concerned with proving a *fixed* hypothesis, say $h(\tilde{x})$, that is, to show that $h()$ holds over all values of the variables $\tilde{x}$. The challenge is to discover and prove $h(\tilde{x}) \implies h(\tilde{x}')$, where expression $\tilde{x}$ is less than the expression $\tilde{x}'$ in some well-founded measure. Furthermore, a *base case* $h(\tilde{x}_0)$ needs to be proven. Automating this form of induction usually relies on the fact that some subset of $\tilde{x}$ are variables of *inductive types*. In contrast, our notion of induction hypothesis is completely different. First, we do not require that some variables are of inductive (and well-founded) types. Second, the induction hypotheses are not supplied explicitly. Instead, they are constructed implicitly via the discovery of a valid proof path. This allows much more potential for automating the proof search. Third (and this also applied to the "Cyclic Proof" method mentioned below), multiple induction hypotheses can be exploited within a single proof path. Without this, as a concrete example, we would not be able to prove $\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) \models \mathsf{ls}(\mathtt{x},\mathtt{y})$.

We further highlight the work of Lahiri and Qadeer [19], which adapts the induction principle for proving properties of well-founded linked list. The technique relies on the well-foundedness of the heap, while employing the 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 simple programs.

We now mention works on "Cyclic Proof", e.g., [7, 8]; and also a somewhat related concept called "Matching Logic" [34]. "Cyclic Proof" replaces explicit induction reasoning by detecting well-founded *infinite descent* over the cyclic proof graphs. (We note that the current implementations of "Cyclic Proof" [7, 8], however, are very limited.) The crucial departure from our work in this paper is that the above-mentioned methods do not deal with the notion of *applying an induction step* in order to *generate* a new and different proof obligation. The power of our methodology comes from the fact that the induction step can be applied repetitively along a proof path, as in the proof of $\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) \models \mathsf{ls}(\mathtt{x},\mathtt{y})$.

We finally mention the work [17], from which the concept of our automatic induction originates. The current paper extends [17] first by refining the original single coinduction rule into two more powerful rules, to deal with the antecedent and consequent of a VC respectively. Secondly, the *application* of the rules has been systematized so as to produce a rigorous proof search strategy. Another technical advance is our introduction of *timestamps* (a progressive measure) in the two induction rules as an efficient technique to avoid circular reasoning. Finally, the present paper focuses on program verification and uses a specific domain of discourse involving the use of explicit symbolic heaps and separation.

7. Concluding Remarks

We presented a framework for proving recursive properties of data structures providing a new level of automation across a wider class of programs. Its key technical feature is the automatic use of induction. More specifically, the framework allows for selecting a dynamically generated proof obligation as an induction hypothesis, and then using this formula in an induction step in order to generate a new proof obligation. The main technical challenge of avoiding circular reasoning was overcome by an intricate restriction on variable renamings. Finally, experimental evidence was presented to show that many real-life proofs, including those of lemmas whose unproved use has been necessary in previous systems, can now be fully automated.

[6] Since the number of rules (disjuncts) in a predicate definition is fixed and usually small, the size of proof tree mainly depends on the number of unfolds and inductions.

[7] We believe that the lemmas in [28] are unnecessarily complicated, because the authors want to reduce the number of them, by grouping a few into one.

References

[1] J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, pages 97—-109, 2004.

[2] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52—68, 2005.

[3] N. Bjørner and J. Hendrix. Linear functional fixed-points. In *CAV, LNCS 5643*, pages 124—139. Springer, 2009.

[4] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. A logic- based framework for reasoning about composite data structures. In *CONCUR, LNCS 5710*, pages 178—195. Springer, 2009.

[5] R. S. Boyer and J. S. Moore. A theorem prover for a computational logic. In *CADE*, 1990.

[6] J. Brotherston and J. Villard. Parametric completeness for separation theories. In *POPL*, pages 453–464, 2014.

[7] J. Brotherston, D. Distefano, and R. L. Petersen. Automated cyclic entailment proofs in separation logic. In *CADE*, 2011.

[8] J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *APLAS*, pages 350–367, 2012.

[9] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. In *SCP*, pages 1006—1036, 2012.

[10] D. H. Chu, J. Jaffar, and M. T. Trinh. Automatic induction proofs of data-structures in imperative programs. Technical report, National University of Singapore, 2015.

[11] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *TPHOLs*, 2009.

[12] P. C. Dillinger, P. Manolios, D. Vroon, and J. S. Moore. ACL2s: "The ACL2 Sedan". In *ICSE*, 2007.

[13] G. Duck, J. Jaffar, and N. Koh. A constraint solver for heaps with separation. In *CP, LNCS 8124*, 2013.

[14] R. Iosif, A. Rogalewicz, and J. Simachek. The tree width of separation logic with recursive definitions. In *CADE*, 2013.

[15] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NFM*, pages 41—55, 2011.

[16] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.

[17] J. Jaffar, A. E. Santosa, and R. Voicu. A coinduction rule for entailment of recursively defined properties. In *CP*, 2008.

[18] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL*, pages 171—182. ACM, 2008.

[19] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126. ACM, 2006.

[20] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR, LNCS 6355*, pages 348—370. Springer, 2010.

[21] K. R. M. Leino. Automating induction with an smt solver. In *VMCAI*, 2012.

[22] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *POPL*, pages 611–622. ACM, 2012.

[23] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Thor: A tool for reasoning about shape and arithmetic. In *CAV*, 2008.

[24] P. Navarro and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, 2011.

[25] H. H. Nguyen and W. N. Chin. Enhancing program verification with lemmas. In *CAV '08*, pages 355–369. Springer-Verlag, 2008.

[26] E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in c using separation logic. In *PLDI*, 2014.

[27] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In *CAV*, 2013.

[28] X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, 2013.

[29] Z. Rakamaric, J. D. Bingham, and A. J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *VMCAI*, 2007.

[30] Z. Rakamaric, R. Bruttomesso, A. J. Hu, and A. Cimatti. Verifying heap-manipulating programs in an smt framework. In *ATVA*, 2007.

[31] S. Ranise and C. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *SEFM*, pages 206—215, 2006.

[32] J. Reynolds. A short course on separation logic. `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/wwwaac2003/aac.html`, 2003.

[33] J. C. Reynolds. Separation logic: A logic for shared mutable data objects. In *LICS*. IEEE, 2002.

[34] G. Rosu and A. Stefanescu. Checking reachability using matching logic. In *OOPSLA '12*, pages 555–574. ACM, 2012.

[35] W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *TACAS*, 2012.

[36] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349—361. ACM, 2008.

[37] K. Zee, V. Kuncak, and M. Rinard. An integrated proof language for imperative programs. In *PLDI*, pages 338—-351. ACM, 2009.