Abstract

We consider the problem of automated reasoning about dynamically manipulated data structures. Essential properties are encoded as predicates whose definitions are formalised 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 from 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. Our main result overcomes this with a novel checking condition. 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 very large search space. In the end, our method automatically reasons about a new class of formulas arising from practical program verification.

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” [Navarro and Rybalchenko 2011]. Informally, in the regard of handling recursive predicates, the state-of-the-art [Chin et al. 2012; Madhusudan et al. 2012; Qiu et al. 2013; Piskac et al. 2013; Pek et al. 2014], 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 \( p(\bar{u}) \wedge \cdots \models p(\bar{v}) \) is the formula, then this is obviously provable if \( \bar{u} \) and \( \bar{v} \) were unifiable (under an appropriate theory governing the meaning of the expressions \( \bar{u} \) and \( \bar{v} \)). In other words, we have performed “formula abstraction” [Madhusudan et al. 2012] by treating the recursively defined term \( p() \) 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 [Berdine et al. 2005]. 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 pred-
icate in the RHS which cannot be transformed, by 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:

\[
\begin{align*}
\text{vlist} (x) & \overset{def}{=} x = \text{null} \land \text{emp} \\
| (x \rightarrow t) & \overset{def}{=} \text{vlist}(t)
\end{align*}
\]

\[
\begin{align*}
\text{zero_list} (x) & \overset{def}{=} x = \text{null} \land \text{emp} \\
| (x \rightarrow 0, t) & \overset{def}{=} \text{zero_list}(t)
\end{align*}
\]

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

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

\[
\begin{align*}
\text{True} & \\
\text{(OBVIOUS)} & (x \rightarrow 0, t) * \text{vlist}(t) \iff (x \rightarrow 0, t) * \text{vlist}(t) \\
\text{(SUBSTITUTION)} & (x \rightarrow 0, t) * \text{vlist}(t) \iff (x \rightarrow 0, z) * \text{vlist}(z) \\
\text{(RIGHT-UNFOLD)} & (x \rightarrow 0, t) * \text{vlist}(t) \iff \text{vlist}(x) \text{ (iii)} \\
\text{(LEFT-WEAKEN)} & (x \rightarrow 0, t) * \text{zero_list}(t) \iff \text{vlist}(x) \text{ (ii)}
\end{align*}
\]

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 as in the previous proof:

\[
\begin{align*}
\text{True} & \\
\text{(OBVIOUS)} & (x \rightarrow \omega, t) * \text{vlist}(t) \iff (x \rightarrow \omega, t) * \text{vlist}(t) \\
\text{(SUBSTITUTION)} & (x \rightarrow \omega, t) * \text{vlist}(t) \iff (x \rightarrow \omega, z) * \text{vlist}(z) \\
\text{(RIGHT-UNFOLD)} & (x \rightarrow \omega, t) * \text{vlist}(t) \iff \text{vlist}(x) \text{ (3)} \\
\text{(RIGHT-STRENGTHEN)} & (x \rightarrow \omega, t) * \text{vlist}(t) \iff \text{zero_list}(x) \text{ (2)}
\end{align*}
\]

Once again, we use the original proof obligation vlist(\(t\)) \(\vdash\) zero_list(\(t\)) as an induction hypothesis, and this time, we transform the proof obligation (2) into (3): strengthen the RHS by replacing zero_list(\(x\)) with vlist(\(x\)). 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. More specifically, the 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 discharge these lemmas.
- 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 unfolding, often induce very 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, and also the recent “Cyclic Proof” approach [Brotherston et al. 2011, 2012]. 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

\(^{1}\)This is as hard as identifying the class where an invariant discovery technique guarantees to work.
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 [Nguyen and Chin 2008; Chin et al. 2012; Madhusudan et al. 2012; Qiu et al. 2013; Brotherston et al. 2012], 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 in imperative languages.

We first highlight scenarios, which are ubiquitous in realistic programs, and often lead to proof obligations involving unmatchable predicates. Later, we discuss the restriction of U+M paradigm in dealing with such proof obligations.

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.

Figure 1: Partial Proof Tree for \( vlist(x) \models vlist(x) \)

```
(OBVIOUS) \[ x = \text{null} \land \text{emp} \models x = \text{null} \land \text{emp} \]
(RIGHT-UNFOLD) \[ x = \text{null} \land \text{emp} \models vlist(x) (i) \]
(LEFT-UNFOLD) \[ (x \rightarrow 0, t) * \text{zero_list}(t) \models vlist(x) (ii) \]
```

Figure 2: Partial Proof Tree for \( vlist(x) \models \text{zero_list}(x) \)

```
(OBVIOUS) \[ x = \text{null} \land \text{emp} \models x = \text{null} \land \text{emp} \]
(RIGHT-UNFOLD) \[ x = \text{null} \land \text{emp} \models \text{zero_list}(x) (1) \]
(LEFT-UNFOLD) \[ (x \rightarrow _-, t) * \text{vlist}(t) \models \text{zero_list}(x) (2) \]
```

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.

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 modally. 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 [Reynolds 2003]. 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 precondition \( \Phi \), calling function \( \text{func}_a \) and then \( \text{func}_b \). We then need to establish the post-condition \( \Psi \). In tradi-

\(\text{def}\) \(\hat{\mathcal{S}}(x, y) \triangleq x = y \land \text{emp} \)
\(\frac{\begin{align*}
\hat{\mathcal{S}}(x, y) \equiv & x = y \land \text{emp} \\
\text{if} & x \neq y \land (x \rightarrow t) & \hat{\mathcal{S}}(t, y) 
\end{align*}}{(c) \text{ List Segment Definition}}\)

Figure 3: Implementation of a Queue

To illustrate, Fig. 3 shows the implementation of a queue using list segments, extracted from the open source pro-

\(\text{def}\) OpenBSD/queue.h. 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.

2Typically, list segment can be defined in two ways: the moving pointer is either the left one or the right one.
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 [Madhusudan et al. 2012; Qiu et al. 2013], we shall call the DRAYDAD 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 rules which define the predicates of interest. For example, assume the definition for list segment is in Fig. 3(c) and the code fragment in Fig. 5(a).

Figure 4: Modular Program Reasoning

The formal 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) 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 [Chin et al. 2012], 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:

\[
\text{list}(x) \overset{def}{=} x = \text{null} \land \exists \text{emp} \quad | 
\quad (x \rightarrow t) * \text{list}(t)
\]

Though \(\hat{\text{ls}}\) and \(\text{list}\) are closely related, U+M can prove neither of the following obligations:

\[
\hat{\text{ls}}(x, \text{null}) \models \text{list}(x) \quad (2.1)
\]
\[
\hat{\text{ls}}(x, y) \cdot \text{list}(y) \models \text{list}(x) \quad (2.2)
\]

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.

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

As stated in Section 1, the dominating technique to manipulate 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.

Figure 5: U+M with List Segments

Here we want to prove that given \(\hat{\text{ls}}(x, y)\) at the beginning, we should have \(\hat{\text{ls}}(z, y)\) at the end. Since the code touches the “footprint” of \(x\) (second statement), it directs the unfolding of the predicate \(\hat{\text{ls}}(x, y)\) containing \(x\), to expose \(x \neq y \land (x \rightarrow t) \cdot \text{ls}(t, y)\). 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 \(\hat{\text{ls}}\) does not change the second argument of the predicate \(\text{ls}\). 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 [Chin et al. 2012; Qiu et al. 2013], they get around the limitation of U+M via the use, without proof, of additional user-provided “lemmas” (the corresponding term used in [Qiu et al. 2013] is “axioms”). As a matter of fact, 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

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 [Reynolds 2003]. [Duck et al. 2013], in this vein, extends Hoare Logic with explicit heaps. This extension allows for strongest post
conditions, and is therefore suitable for “practical program verification” [Brotherston and Villard 2014] via constraint-based symbolic execution.

In this paper, we start with the existing specification language in [Duck et al. 2013], 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 [Duck et al. 2013] for more details. A heap is a finite partial map from positive integers to integers, i.e. $H = \mathbb{Z}_+ \rightarrow_{hn} \mathbb{Z}$. Given a heap $h \in H$s with domain $D = \text{dom}(h)$, we sometimes treat $h$ as the set of pairs $\{(p, v) \mid p \in D \land v = h(p)\}$. We note that when a pair $(p, v)$ belongs to some heap $h$, it is necessary that $p$ is not a null pointer, i.e., $p \neq 0$. The $\mathcal{H}$-language is the first-order language over heaps.

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

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

As in [Duck et al. 2013], $\mathcal{H}$ is then extended with user-defined recursive predicates. We use the framework of Constraint Logic Programming (CLP) [Jaffar and Maher 1994] to inherit its syntax, semantics, and most importantly, its built-in notions of unfolding rules. For the sake of brevity, we just informally explain the language. The following rules constitutes a recursive definition of predicate list($x, L$), which specifies a list.

\[
\begin{align*}
\text{list}(x, L) & \;\vdash \; x = 0, \; L \equiv \Omega. \\
\text{list}(x, L) & \;\vdash \; L \equiv (x \ast t) \ast L_1, \; \text{list}(t, L_1).
\end{align*}
\]

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 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 list is the empty heap, the root node is equal to null ($x = 0$), and the heap is empty ($L \equiv \Omega$). Otherwise, we can split the heap $L$ into two disjoint parts: a singleton heap $(x \ast t)$ 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.

\[
\begin{align*}
\text{ls} \,(x, y, L) & \;\vdash \; x = y, \; L \equiv \Omega. \\
\text{ls} \,(x, y, L) & \;\vdash \; x \neq y, \; L \equiv (x \ast t) \ast L_1, \; \text{ls}(t, y, L_1). \\
\text{ls} \,(x, y, L) & \;\vdash \; x = y, \; L \equiv \Omega. \\
\text{ls} \,(x, y, L) & \;\vdash \; x \neq y, \; L \equiv (t \ast y) \ast L_1, \; \text{ls}(x, t, L_1).
\end{align*}
\]

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 [Duck et al. 2013]. Specifically, in order to prove the Hoare triple $\{ \phi \} S\{ \psi \}$ for a loop-free program $S$, we simply generate strongest postcondition $\psi’$ 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 and the symbolic execution rules of [Duck et al. 2013].

4. The Proof Method

Background on CLP: This is provided for the convenience of the readers. An atom is of the form $p(\vec{t})$ where $p$ is a user-defined predicate symbol and $\vec{t}$ is a tuple of $\mathcal{H}$ terms. A rule is of the form $A : - \Sigma, B$ where the atom $A$ is the head of the rule, and the sequence of atoms $B$ and the constraint $\Sigma$ 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 $\theta$ simultaneously replaces each variable in a term or constraint $\epsilon$ into some expression, and we write $\epsilon[\theta]$ 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 $\Sigma$ is a constraint, a grounding of $\Sigma$ results in true or false in the usual way.

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

Let $G \equiv \{ B_1, \cdots, B_m, \Sigma \}$ and $P$ denote a non-final goal and a set of rules respectively. Let $R \equiv A : - \Sigma_1, C_1, \cdots, C_m$ denote a rule in $P$, written so that none of its variables appear in $G$. Let the equation $A = B$ be shorthand for the pairwise equation of the corresponding arguments of $A$ and $B$. A reduct of $G$ using a clause $R$, denoted reduct($G, R$), is of the form

$(B_1, \cdots, B_{i-1}, C_1, \cdots, C_m, B_{i+1}, \cdots, B_n, B_i \equiv A, \Sigma, \Sigma_1)$ provided the constraint $B_i = A \land \Sigma \land \Sigma_1$ is satisfiable.
A derivation sequence for a goal \( G_0 \) is a possibly infinite sequence of goals \( G_0, G_1, \ldots \), where \( G_i, i > 0 \) is a reduct of \( G_{i-1} \). A derivation tree for a goal is defined in the obvious way.

**Definition 1 (Unfold).** Given a program \( P \) and a goal \( G \): UNFOLD \( G \) is \( \{G' | \exists R \in P : G' = \text{reduct}(G, R)\} \).

Given a goal \( L \) and an atom \( p \in L \), UNFOLD\( P \)(\( L \)) denotes the set of formulas transformed from \( L \) by unfolding \( p \).

**Definition 2 (Entailment).** An entailment is of the form \( L \vdash R \), where \( L \) and \( R \) are goals.

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

### 4.1 Unfold and Match (U+M)

Assume that we start off with \( L \vdash R \). 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 \( p \in L \), unfold \( L \) wrt. \( p \) and obtain the goals \( L_1, \ldots, L_n \). The validity of the original entailment can now be obtained by ensuring the validity of \textit{all} the entailments \( L_i \vdash R \) (\( 1 \leq i \leq n \)).

- Second, select a recursive atom \( q \in R \), unfold \( R \) wrt. \( q \) and obtain the goals \( R_1, \ldots, R_m \). The validity of the original entailment can now be obtained by ensuring the validity of \textit{any one} of the entailments \( L \vdash R_j \) (\( 1 \leq j \leq m \)).

So the proof process can proceed recursively either by proving \textit{all} \( L_i \vdash R \) or by proving \textit{one} \( L \vdash R_j \) 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, such tree size is often manageable.

### 4.2 Formula Re-writing with Dynamic Induction Hypotheses

We now present a formal calculus for the proof of \( L \vdash R \) 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 \( \tilde{A} \vdash L \implies R \) where \( L \) and \( R \) 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.

The role of proof obligations is to capture the state of the proof process. Each element in \( A \) 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 \( L \vdash R \), our proof shall start with \( \emptyset \vdash L \implies R \), 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, \( L \vdash_{\text{SMT}} R \) denotes the validity of \( L \vdash R \) is obtained simply by consulting a generic SMT solver.

- The substitution (SUB) rule removes one occurrence of an assertion predicate, say atom \( p(y) \), 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 \textit{direct proof}. In principle, it is similar to the process of “matching” in the U+M paradigm. For brevity we then use \( L \vdash_{\text{DP}} R \) to denote the fact that the validity of \( L \vdash R \) can be proved directly using only (SUB) and (CP) rules.

- The \textit{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 \( L \vdash R \) as an assumption, but also need to keep track of the atom \( p \). This is why in the rule we see a pair \( \langle L \vdash R; p \rangle \) added into the current set of assumptions \( \tilde{A} \).

On the other hand, the \textit{right unfold} (RU) rule selects some recursive atom \( q \) and performs an unfold on the RHS of a
SUB to treat previously encountered obligations as possible

two rules, also called the “induction rules” for short, allow

variable

In Fig. 7, we show how this proof obligation can be suc-
applied by dispensing with (SUB), (RU), and (CP) rules

• 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+1) 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 \( L \models R \),
we now proceed by finding \( \overline{L} \) and \( \overline{R} \) such that \( L \models \overline{L} \)
\( \models \overline{R} \). The key here is to find those candidate goals where the
validity of \( \overline{L} \models \overline{R} \) directly follows from a “similar”
assumption \( A \), together with \( \theta \) to rename all the variables in
\( A \) to the variables in the current obligation, namely \( L \models R \).

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
\( \overline{L} \) and \( \overline{R} \) so we can (easily) find a renaming \( \theta \) such that
\( A \theta \Rightarrow \overline{L} \models \overline{R} \) ( \( \Rightarrow \) denotes logical implication).

To be more deterministic and to prevent us from trans-
moving to obligations harder than the original one, we re-

quire that at least one of the remaining two entailments,

\( L \models \overline{L} \) and \( \overline{R} \models R \), is discharged quickly by a
direct proof.

In (IA-1) rule, given the current obligation \( p(\bar{x}) \land L_1 \land
L_2 \models R \) and an assumption \( A \equiv p(\bar{y}) \land L' \models \overline{R}' \), we
choose \( p(\bar{x}) \land L' \land L_2 \) to be our \( \overline{L} \) and \( \overline{R}' \land L_2 \) to be our
\( \overline{R} \). We can see that the validity of \( \overline{L} \models \overline{R} \) directly follows
from the assumption \( A \theta \). One restriction onto the renaming
\( \theta \), to avoid circular reasoning, is that \( \theta \) must rename \( \bar{y} \) to \( \bar{x} \)
where \( p(\bar{x}) \) is an atom which has been generated after \( p(\bar{y}) 

\) had been unfolded. Such fact is indicated by \( \text{gen}(p(\bar{x})) \geq
\text{kill}(p(\bar{y})) \) in our rule. While \( \text{gen}(p) \) denotes the timestamp
when the recursive atom \( p \) is generated during the proof
process, \( \text{kill}(p) \) denotes the timestamp when \( p \) is unfolded
and removed. Another side condition for this rule is that the
validity of \( \overline{L} \models \overline{R} \), or equivalently, \( L_1 \equiv \overline{L} \theta \)
is discharged immediately by a direct proof.

In (IA-2) rule, given the current obligation \( p(\bar{x}) \land L_1 \models
R \) and an assumption \( A \equiv p(\bar{y}) \land L' \models R' \), on the other hand,
\( p(\bar{y}) \land \overline{L}' \theta \) serves as our \( \overline{L} \) while \( \overline{R}' \theta \) serves as our \( \overline{R} \).

The validity of \( \overline{L} \models \overline{R} \) trivially follows from the assumption \( A \theta 

\) namely \( p(\bar{x}) \land \overline{L} \models \overline{R} \theta \). As in (IA-1), we also put similar
restriction upon the renaming \( \theta \). Another side condition
we require is that the validity of \( \overline{L} \models \overline{R} \) 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 \( \theta \). Here we make sure that \( \theta \) renames
atom \( p(\bar{y}) \) to \( p(\bar{x}) \), where \( p(\bar{x}) \) has been generated after
\( p(\bar{y}) \) had been unfolded (and removed). This helps to rule out
certain potential \( \theta \) 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

\begin{align*}
\text{(CP)} & \quad \frac{\text{True}}{A \models \overline{L} \models R} \quad \text{where recursive atoms are treated as uninterpreted} \\
\text{(SUB)} & \quad \frac{\overline{A} \models \overline{L} \land \overline{p}(\bar{x}) \models \overline{R} \theta}{A \models \overline{L} \land \overline{p}(\bar{x}) \models \overline{R}} \quad \text{there exists a substitution } \theta \text{ for existential variables in } \bar{y} \text{ s.t. } \overline{L} \land \overline{p}(\bar{x}) \models \overline{R} \theta = \bar{y} \theta \\
\text{(LU+1)} & \quad \frac{\bigcup_{i=1}^{n} \{ \overline{A} \cup \{ \overline{L} \models \overline{R} ; p \} \} \models \overline{L_i} \models \overline{R}}{A \models \overline{L} \models \overline{R}} \quad \text{Select an atom } p \in \overline{L} \text{ and UNFOLD}_p(\overline{L}) = \{ L_1, \ldots, L_n \} \\
\text{(RU)} & \quad \frac{\overline{A} \models \overline{R'} \land \overline{L_2} \models \overline{R}}{A \models \overline{L} \models \overline{R'}} \quad \text{Select an atom } \overline{q} \in \overline{R} \text{ and } \overline{R'} \subseteq \text{UNFOLD}_q(\overline{R}) \\
\text{(IA-1)} & \quad \frac{\overline{A} \models \overline{R'} \land \overline{L_2} \models \overline{R}}{A \models \overline{p}(\bar{x)} \land \overline{L_1} \land \overline{L_2} \models \overline{R}} \quad (p(\bar{y}) \land \overline{R'} ; p(\bar{y})) \in \overline{A} \text{ and } \text{gen}(p(\bar{x})) \geq \text{kill}(p(\bar{y})), \text{there exists a renaming } \theta \text{ s.t. } \bar{x} = \bar{y} \theta \text{ and } \overline{L_1} \models \overline{L} \theta \\
\text{(IA-2)} & \quad \frac{\overline{A} \models \overline{L_1} \models \overline{L'} \theta}{A \models \overline{p}(\bar{x}) \land \overline{L_1} \models \overline{R}} \quad (p(\bar{y}) \land \overline{L'} ; p(\bar{y})) \in \overline{A} \text{ and } \text{gen}(p(\bar{x})) \geq \text{kill}(p(\bar{y})) \text{ and there exists a renaming } \theta \text{ s.t. } \bar{x} = \bar{y} \theta \text{ and } \overline{L'} \theta \models \overline{R} \\
\end{align*}
proof for \( A \) might not be valid. This is the reason why for each element of \( \bar{A} \), we not only keep track of the assumption, but also the recursive atom \( p \) to which the application of \((LU+1)\) 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 \( \theta \). In other words, we constrain the use of the rules, which is transparent to the user, in order 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. The proof of soundness can be found in our supplementary. Consider the recursive predicate \( p \), defined as

\[ p(x) ::= p(x). \]

and the following two proof obligations:

\[ p(x) \vdash \text{list}(x, L) \quad (4.1) \]
\[ \text{list}(x, L) \vdash p(x) \quad (4.2) \]

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\(^3\). On the other hand, (4.2) does not hold because \( x = 0 \) (and \( L \equiv \Omega \)) is a model of the LHS, but not a model of the RHS.

![Figure 8: Our Proof for (4.1)](image)

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

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

![Figure 9: An Unsuccessful Attempt for (4.2)](image)

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

\[ \{ A' \vdash x = 0, L \equiv \Omega \vdash p(x) \} \text{ and} \]
\[ \{ A' \vdash L \equiv (x \mapsto t) + L_1, \text{list}(t, L_1) \vdash p(x) \}, \]

where \( A' \equiv \{ \text{list}(x, L) \vdash p(x); \text{list}(x, L) \} \).

Focus on the obligation \( \{ A' \vdash x = 0, L \equiv \Omega \vdash p(x) \} \). Clearly consulting a constraint solver or performing substitution does not help. Rule \((LU+1)\) is not applicable since no recursive predicate on the LHS. As before, we cannot progress using \((RU)\) rule. Importantly, the side conditions prevent (1A-1) and (1A-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: \( \text{ls}(x, y, L) \vdash \text{ls}(x, y, L) \).

Our method can discharge this obligation by applying (1A-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 \( \text{ls}(x, y, L) \) in the LHS of the given obligation via \((LU+1)\) 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 rewriting step, i.e., an application of (1A-1) rule. Similarly, in the third step, we unfold the predicate \( \text{ls}(t, y, L_1) \) in the LHS via \((LU+1)\) 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 (1A-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., (CP) rule.

**Example 3.** Consider the entailment:

\[ \text{ls}(x, y, L_1), \text{list}(y, L_2), L_1 \uplus L_2 \vdash \text{list}(x, L), L \equiv L_1 \uplus L_2. \]

Fig. 11 shows, only the interesting proof path, how we can successfully prove this entailment using the (1A-2) rule. We first unfold \( \text{ls}(x, y, L_1) \) in the LHS, adding \( A \) into the set of assumptions. Then using \( A \) as an induction hypothesis, we can rewrite the current obligation via (1A-2) rule. Note that, here we use (1A-2) rule instead of (1A-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., (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 (1A-2)
we now attempt to apply rule (IA-1). The requirement for \( \theta \) forces it to rename \( x \) to \( x \) and \( y \) to \( t \). However, the side condition \( L_1 \models \varphi \ L(\theta) \) cannot be fulfilled, since \( x \neq y, L_1 \equiv (t \rightarrow y) \ast L_3, \text{list}(y, L_2_1), L_1 \ast L_2 \equiv \varphi \ \text{list}(t, \_). \)

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 \( \theta \) to rename \( x \) back to \( x \). On the LHS, we further require \( \theta \) to rename \( y \) to \( t \) so that \( \text{ls}(x, t) \equiv \text{ls}(x, y) \theta \). Note that \( \text{ls}(x, t) \) was indeed generated after \( \text{ls}(x, y) \) had been unfolded and removed (i.e., killed). The remaining transformation is straightforward.

5. Experiments

Our evaluations are performed on a 3.2Gz Intel processor with 2GB RAM, running Linux. We evaluated our prototype on a comprehensive set of benchmarks, including both academic algorithms and real programs. The benchmarks are collected from existing systems [Nguyen and Chin 2008; Chin et al. 2012; Madhusudan et al. 2012; Qiu et al. 2013; Brotherston et al. 2012], those considered as the state-of-the-art for the purpose of proving user-defined recursive data-structure properties in imperative languages. 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 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 [Chin et al. 2012; Madhusudan et al. 2012; Qiu et al. 2013]. As expected, our prototype proves all of those obligations, the running time for each is negligible (less than 0.2 second). This is because the proof obligations usually require just either one left unfold or one right unfold before matching – i.e., a direct proof – can successfully take place. The second set of benchmarks are from “Cyclic Proof” [Brotherston et al. 2012], 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 [Nguyen and Chin 2008; Chin et al. 2012; Madhusudan et al. 2012; Qiu et al. 2013], 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 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

See https://github.com/mihasighi/smtcomp14-sl
Logic syntax for succinctness. Note that due to the duality of the definitions for list segments, e.g., $ls$ vs. $\hat{ls}$, each lemma containing them would usually have 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).

<table>
<thead>
<tr>
<th>Lemma</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sorted_list(x, min) \models list(x)$</td>
</tr>
<tr>
<td>$sorted_list(x, len, min) \models list(x, len)$</td>
</tr>
<tr>
<td>$sorted_list(x, len, min) \models sorted_list(x, min)$</td>
</tr>
<tr>
<td>$sorted_ls(x, y, min, max) \models sorted_list(y, min, 2)$</td>
</tr>
<tr>
<td>$\land \ max \leq min_2 \models sorted_list(x, min)$</td>
</tr>
<tr>
<td>$ls(x, y) \models list(y) \models list(x)$</td>
</tr>
<tr>
<td>$ls(x, y) \models \hat{ls}(x, y)$ and $\hat{ls}(x, y) \models ls(x, y)$</td>
</tr>
<tr>
<td>$\hat{ls}(x, y, len_1) \models \hat{ls}(y, z, len_2) \models \hat{ls}(x, z, len_1 + len_2)$</td>
</tr>
<tr>
<td>$ls(x, y, len_1) \models \hat{ls}(y, len_2) \models ls(x, len_1 + len_2)$</td>
</tr>
<tr>
<td>$ls(x, last, len) \models \hat{ls}(x, new, len + 1)$</td>
</tr>
<tr>
<td>$dls(x, y) \models \hat{dlist}(y) \models dlist(x)$</td>
</tr>
<tr>
<td>$d\hat{ls}_1(x, y, len_1) \models \hat{dlist}(y, z, len_2) \models \hat{dls}_1(x, z, len_1 + len_2)$</td>
</tr>
<tr>
<td>$d\hat{ls}_2(x, y, len_1) \models \hat{dlist}(y, len_2) \models d\hat{ls}_2(x, len_1 + len_2)$</td>
</tr>
<tr>
<td>$avl(x, hgt, min, max, balance) \models bstree(x, hgt, min, max)$</td>
</tr>
<tr>
<td>$bstree(x, height, min, max) \models bintree(x, height)$</td>
</tr>
</tbody>
</table>

Let us briefly discuss 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 very large search space.

We now use the set of academic algorithms and open-source library programs, collected and published by [Chin et al. 2012; Qiu et al. 2013], to demonstrate that our prototype can verify all of the programs in this set without using 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).

<table>
<thead>
<tr>
<th>Program</th>
<th>Function</th>
<th>T/F</th>
</tr>
</thead>
<tbody>
<tr>
<td>glib/glist.c</td>
<td>find, position, index, nth, last, length, append, insert_at_pos, merge_sort, remove, insert_sorted_list</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>glib/gliblist.c</td>
<td>nth, position, find, index, last, length</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>queue.h</td>
<td>simpleq.remove_after, simpleq.insert_tail, simpleq.insert_after</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>ExpressOS/cachePage.c</td>
<td>lookup_prev, add_cachepage</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>linux/mmmap.c</td>
<td>insert_vm_struct</td>
<td>&lt;1s</td>
</tr>
</tbody>
</table>

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

Remark #2: The verification time for each function is always less than 1 second. This is within our expectation since when 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/glist.c, we only need to prove 3 obligations, each of which requires no more than two left unfolds, two right unfolds and two inductions. In fact, the maximum number of left unfolds, right unfolds and inductions used in our system are 5, 5

5Since the number of rules (disjuncts) in a predicate definition is fixed and small, the size of proof tree mainly depends on the number of unfolds and inductions.
and 3 respectively, even for the functions that take U+M frameworks much longer time to prove. For example, consider \texttt{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) [Reynolds 2002] is often combined with a recursive formulation of data structure properties. Implementations, however, are incomplete, e.g., [Berdine et al. 2005; Iosif et al. 2013], or deal only with fragments [Berdine et al. 2004; Magill et al. 2008]. There is also literature on decision procedures for restricted heap logics; we mention just a few examples: [Rakamaric et al. 2007a,b; Lahiri and Qadeer 2008; Ranise and Zarba 2006; Bouajjani et al. 2009; Bjørner and Hendrix 2009]. 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 [Leino 2010], VCC [Cohen et al. 2009] and Verifast [Jacobs et al. 2011] which require significant ghost annotations, and annotations that explicitly express and manipulate frames. They do not automatically verify the general and complex properties addressed in this paper, but in general resort to interactive theorem provers, e.g. Mona, Isabelle or Coq, which usually requires manual guidance.

[Navarro and Rybalchenko 2011] showed that significant performance improvements can be obtained by incorporating first-order theorem proving techniques into SL provers. However, the focus of that work is about list segments, not general user-defined recursive predicates. On a similar thread, [Piskac et al. 2013] advances the automation of SL, using SMT, in verifying procedures manipulating list-like data structures. The works [Zee et al. 2008, 2009; Chin et al. 2012; Madhusudan et al. 2012; Qiu et al. 2013] are also close related works: 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 [Boyer and Moore 1990; Dillinger et al. 2007; Leino 2012; Sonnex et al. 2012]. They are concerned with proving a fixed hypothesis, say \( h(\bar{x}) \), that is, to show that \( h() \) holds over all values of the variables \( \bar{x} \). The challenge is to discover and prove \( h(\bar{x}) \implies h(\bar{x}') \), where expression \( \bar{x}' \) is less than the expression \( \bar{x} \) in some well-founded measure. Furthermore, a base case of \( h(\bar{x}_0) \) needs to be proven. Automating this form of induction usually relies on the fact that some subset of \( \bar{x} \) is variables of \textit{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 \( \text{ls}(x,y) \Rightarrow \text{ls}(x,y) \).

We further highlight the work of Lahiri and Qadeer [Lahiri and Qadeer 2006], 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” [Brotherston et al. 2011, 2012] and “Matching Logic” [Rosu and Stefanescu 2012]. They are based on the principle that when two similar obligations are detected in the same proof path, the latter can be ignored, or considered proved. The effect of this is that if a proof depends on two subproofs and one of them is cyclic, then the proof can proceed solely on the second subproof. The crucial departure from our work in this paper is that the Cyclic Proof methodology does not deal with the notion of applying an induction step in order to generate a new and different proof obligation. It only allows the conclusion of one proof sequence so that others can proceed and if these others succeed, it provides termination. To be concrete, it cannot deal with the initial motivating example about zero lists in Section 1.

We finally mention the work [Jaffar et al. 2008], from which the concept of our automatic induction originates. The current paper extends [Jaffar et al. 2008] 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 \textit{times-tamps} (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.
References