Proving Data Structure Properties by Automatic Induction

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

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

Minh-Thai Trinh  
National University of Singapore  
thaitm52@gmail.com

Abstract
We consider the problem of automated program verification with emphasis on reasoning about dynamically manipulated data structures. Presently, the state-of-the-art methods are limited to the unfold-and-match (U+M) paradigm where predicates are transformed by fold/unfold operations induced from their recursive definitions. A crucial limitation of U+M is that it cannot in general prove properties between different predicates.

Our contribution is a method which can automatically detect and employ induction hypothesis in the proof process, thereby providing a systematic and general method for reasoning about different predicates for the first time. After arguing that the need for this is in fact widespread in practice, we finally demonstrate our method experimentally.

1. Introduction
We consider the problem of automated program verification with emphasis on reasoning about dynamically manipulated data structures. In automated verification, pre/post conditions are typically specified for each method/procedure (and an invariant given for each loop) before the reasoning system automatically checks if each given program code is correct with respect to the given pre/post/invariant annotations. The verification of properties of the dynamically modified heap is a big challenge for logical methods. This is because typical correctness properties of heaps require complex combinations of structure, data, and separation. This paper presents a proof method for general properties of structure, data and separation. Our framework is not restricted to any decidable class and instead attempts to go beyond the classes covered by present methods.

The State-of-the-Art: Unfold and Match
Automated proof of heap properties – usually formalized using separation logic and extended with recursively defined shape predicates – “relies on decidable sub-classes together with the corresponding proof systems based on folding/unfolding strategies for recursive shape definitions” [19]. Informally, the state-of-the-art [5, 21, 20] can be classified as “unfold-and-match” (U+M) techniques. The proof search proceeds by repetitively applying (un)folding strategies, until all the reduced predicates in the RHS of the obligation can be canceled out by corresponding predicates in the LHS via simple, non-recursive, matching method.

More precisely, the only way to manipulate predicates which are user-defined by means of recursive rules is, presently, to employ the basic transformation steps of folding and/or unfolding the rules. What is desired is that a particular sequence of successive applications of these steps produces a final formula which is obviously provable. This usually means that either 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 that no special consideration is needed on any occurrence of a predicate appearing in the final formula. For example, if \( p(\bar{a}) \land \cdots \land p(\bar{v}) \) is the formula, then this is obviously provable if \( \bar{a} \) and \( \bar{v} \) were unifiable (under an appropriate theory governing the meaning of the expressions \( \bar{a} \) and \( \bar{v} \)).

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 [17, 21], 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 operations of the recursive rules which define the predicates of interest. For example, consider the code fragment in Fig. 1(a) and a possible recursive definition for list segment as below:

\[
\text{list}_\text{seg}(x, y) \equiv x = y \land \text{emp} \lor x \neq \text{null} \land t = x.\text{next} \cdot \text{list}_\text{seg}(t, y)
\]

Here we want to prove that given \( \text{list}_\text{seg}(x, y) \) at the beginning, we should have \( \text{list}_\text{seg}(z, y) \) at the end. Since the code touches the “footprint” of \( x \) (second statement), it directs the unfolding of the predicate containing \( x \), namely, \( \text{list}_\text{seg}(x, y) \), to expose \( x \neq \text{null} \land t = x.\text{next} \cdot \text{list}_\text{seg}(t, y) \). The consequent then can be established via a simple matching from variable \( z \) to \( t \).

Limitations of Unfold and Match
The main limitation of U+M is simply that it generally cannot prove a relationship between predicates. We now highlight two distinct scenarios where this phenomenon happens in realistic programs. We first articulate these briefly, and then proceed with examples.

Recursion Divergence
when the “recursion” in the recursive rules is structurally dissimilar to the program code.

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

First, consider recursion divergence. This happens often when the predicates are not unary, i.e., they relate two or more pointer vari-

Figure 1: U+M with List Segments

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>list\text{seg}(x, y)</td>
<td>list\text{seg}(x, y)</td>
</tr>
<tr>
<td>\text{assume}(x \neq \text{null})</td>
<td>\text{assume}(y \neq \text{null})</td>
</tr>
<tr>
<td>\text{z} = x.\text{next}</td>
<td>\text{z} = y.\text{next}</td>
</tr>
<tr>
<td>list\text{seg}(z, y)</td>
<td>list\text{seg}(x, z)</td>
</tr>
</tbody>
</table>

(a) Program 1  
(b) Program 2

[Copyright notice will appear here once ‘preprint’ option is removed.]

[2013/11/28]
Recall the list segment example above. Now we consider a variant code fragment in Fig. 1(b): instead of moving one position away from \( x \), we move one step away from \( y \). To see that U+M, however, cannot work, it suffices to see that unfolding/folding of the list\( \_seg \) predicate does not change the second argument of the predicate. In particular, 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 above ultimately is about how to relate two possible definitions of list segment (recursing either on the left or on the right pointer). One might argue that we can rewrite list segment accordingly to get around the issue. That, however, is not always acceptable. See the implementation of a queue in Fig. 2, extracted from the open source program openbsd/queue.h. Rewriting will not work, because now, we need to cater for both operations: (1) adding a new element into the end of a queue (enqueue); (2) deleting an element at the beginning of it (dequeue). In the two use cases, the “moving pointers” are necessary to recurse differently.

Next, we move along to the scenario where we need to have different predicates, and we need to prove a relationship between them. A particularly important relationship is simply that one is more general than the other. Here obviously unfolding/folding will not be able to reduce the predicates to matchable sets of predicates. Such scenarios are typical when we deal with iterative loops, whose invariants must be consistent with the code, and yet these invariants are only used later in the program to prove a property not specified using the identical predicates of the invariant. We next elaborate.

In iterative algorithms, often the loop invariants cannot be specified using only predicates appearing in the pre- and post-condition of the specifications. For example, programs manipulate lists usually have loops of which the invariants need to talk about list segments. Though list\( \_seg \) and list are closely related, using U+M only, it is impossible to prove list\( \_seg(x, null) \Rightarrow \text{list}(x) \). In order not to give the impression that this generalization problem applies only to the “handles” of data structures, consider the program fragment below:

\[
\begin{align*}
i & = 0; \\
\text{while } (i < n) \{ & a[i] = 1; i++; \\
& i = 0; \\
\text{while } (i < n) \{ & a[i] = a[i] + j; i++; \\
\end{align*}
\]

An invariant for the first loop will generally say that all elements in array \( a[] \) (from indices 0 through \( n - 1 \)) are somewhat related to 1. However, to prove that all the elements are in fact greater than \( j \) at the end, we need a different invariant for the second loop. And, importantly, we need to prove that the first invariant together with the assignment to \( j \) implies the second invariant.

More generally, in typical program development, code reuse is often desired. Therefore, the specification of a function should be independent of the context where it is plugged in. In each context, we might want to establish arbitrarily different properties, of course as long as they are weaker than what the function can guarantee. Without the power to relate different predicates, compositional reasoning is seriously hampered. Fig. 3 gives a concrete example. Here, the function of interest is \text{zero\_list}. It takes in an input, constructs and returns a list of that size, and importantly the data field of each node is initialized to 0. In one place we want to prove that the produced list contains all non-negative data. In another place, however, we want to prove that all the nodes in the produced list share the same data value.

\[
\begin{align*}
\text{true} & \quad x = \text{zero\_list}(a) \\
\text{true} & \quad y = \text{zero\_list}(b) \\
\text{non\_negative}(x) & \quad \text{all\_same}(y)
\end{align*}
\]

\begin{enumerate}
\item Use Case 1
\item Use Case 2
\end{enumerate}

**Figure 2:** Implementation of a queue

\begin{tabular}{|l|l|}
\hline
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 \\
\hline
\end{tabular}

(a) Insert Tail (b) Remove Head

**Figure 3:** Functions Used in Different Contexts

**On using Axioms and Lemmas**

We have argued that the limitations of U+M is severe, mainly because there is a general need for reasoning about different predicates. Therefore there cannot be significant class of programs, including academic programs, which is automatically provable. However, in the literature, there are in fact existing systems displaying proofs of significant examples.

The main reason for this is that existing methods allow properties to be constructed from a predefined set of recursive predicates [20]. Hard-wired rules can then be used to facilitate unfold-and-match. For systems that support general user-defined predicates [5, 21], they get around the limitations of unfold-and-match via the use, without proof, of additional user-provided “lemmas”, in the case of [5], and user-provided “axioms”, in the case of [21]. The general idea is twofold: that

- these formulas have proofs, though manual, that are simple;
- these formulas are general and the number of needed formulas is small.

The axioms used by [21], for example, do not satisfy the two conditions. Some of these axioms are not so obvious (therefore not so reasonable to accept as proven) and some are specifically tailored for the proofs of the target programs (e.g., the axiom used to prove delete\_iter method in a binary search tree).

Nevertheless, the fact of the matter is that there are too many properties that could be accommodated by axioms/lemmas. In existing benchmarks, most of the properties covered essentially concerned the “shape” of the data structures. In practice, what is often needed are custom predicates for specific application domains, and these will involve properties of the data values in addition to shapes. It is thus unacceptable that in order to prove more such programs, we constantly add in more custom axioms/lemmas to facilitate the proof system.

**Our Contribution**

In this paper, we present a framework which in one hand is built upon U+M and therefore subsumes the power of existing unfold/fold reasoners. We begin with an existing specification language which is expressive enough to facilitate the process of automatic program verification where each loop is associated by an invariant. This language is borrowed from [8], 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. We remark here that, most specifications written in traditional languages, e.g., separation logic, can be automatically compiled into this language.

Importantly, however, our framework possesses the key technical feature: automatic induction. This enables us to extend significantly the zone covered by the U+M paradigm. Our use of induction is special and different from its conventional use in theorem proving. There, having decided to apply the induction tactic to an
obligation, the system then start searching for appropriate induction variable(s) with a well-founded measure and appropriate induction hypotheses. Here, in the process of U+M, i.e. systematically unfolding with the hope to reduce the original obligation to some provable formulas, we look for closely similar formulas in one path, where the truth of one would subsume the truth of the other. When such ancestor-descendant pair is found, that proof path can be terminated. This is akin to the process of finding a fix-point in order to terminate the reasoning of an unbounded loop in the setting of traditional program analysis.

The key result is that we now can prove relationships between (different) predicates (of arbitrary arity). We demonstrate, using real programs, that with the power of automatic induction, there is no longer the need for unjustified use of user-provided lemmas/axioms. One direct implication of this is that we have closed the gaps left by many existing works. The potential of induction, however, is not limited to this purpose. Indeed, the concept enables unbounded number of automatic and justified abstraction steps in the search process; therefore we believe our work not only provides a new level of automation for program verification but also can act as a basis for an analysis framework, which we will consider as our future work.

2. The Assertion Language

This section first overviews the syntax and semantics of a language of heaps with separation, which we denote by $H$, that encodes some of the logical connectives of Separation Logic. Essentially, $H$ is a structure modeling a heap, defined as a finite map of positive integers to integers, and integer constraints. Recursive definitions are provided the constraint $θ$ having no variables. A grounding of a goal $P$ is the set of pairs $(p, v)$ where $p$ is an atom, $v$ is a valuation. The constraint $θ$ is the body of the rule, and the sequence of atoms $B$ and the constraint $Ψ$ constitute the body of the rule. The constraint $Ψ$ is a $H$ predicate. 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 substitution $θ$ simultaneously replaces each variable in a term or constraint $e$ into some expression, and we write $eθ$ 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 integer or heap variable into its intended universe of discourse: an integer or an heap. Where $Ψ$ is a constraint, a grounding of $Ψ$ results in true or false in the usual way.

A grounding of an atom $p(\overline{t})$ is an object of the form $p(\overline{t}θ)$ having no variables. A grounding of a goal $G ≡ (p(\overline{t}), Ψ)$ is a grounding of $p(\overline{t})$ where $Ψθ$ is true. We write $[G]θ$ to denote the set of groundings of $G$.

Let $G ≡ (B_1, \ldots , B_n, Ψ)$ and $P$ denote a non-final goal and a set of rules respectively. Let $R ≡ A; -Ψ_1, C_1, \ldots , C_m$, 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 $\text{reduct}(G, R)$, is of the form

$$(B_1, \ldots , B_{i−1}, C_1, \ldots , C_m, B_{i+1}, \ldots , B_n, B_i = A, Ψ, Ψ_1)$$

provided the constraint $B_i = A ∧ Ψ ∧ Ψ_1$ 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 3 (Unfold). Given a program $P$ and a goal $G$, $\text{unfold}(G) = \{G′|∃R ∈ P : G′ = \text{reduct}(G, R)\}$.

The semantics of a set of rules is traditionally known as the “least model” semantics. Essentially, this is the set of groundings of the

A valuation $s$ (a.k.a. variable assignment) is a function mapping integers to $\mathbb{Z} ∪ \text{Heaps}$. We define the semantics of the $H$-language as follows:

Definition 2 (Heap Interpretation). Given a valuation $s$, the $H$-interpretation $I$ is a $\Sigma_H$-interpretation such that:

- $I(v, s) = s(v)$, where $v$ is a variable;
- $I(Ω, s) = \{\}$ (as a Heap);
- $I(p \rightarrow v, s) = \{(q, w)\}$ where $q = I(p, s)$ and $w = I(v, s)$;
- $I(H_1 * \ldots * H_i \models H_{i+1} * \ldots * H_n, s) = \text{true}$ iff for $h_i = I(H_i, s)$ we have that:
  1. $\text{dom}(h_1) \cap \ldots \cap \text{dom}(h_i) = \emptyset$ and $\text{dom}(h_{i+1}) \cap \ldots \cap \text{dom}(h_n) = \emptyset$;
  2. $h_1 \cup \ldots \cup h_i = h_{i+1} \cup \ldots \cup h_n$

Note that we treat each configuration of $⟨∗⟩$ and $≡$ as a distinct predicate. Intuitively, a constraint like $H ≡ H_1 * H_2$ treats $⟨∗⟩$ in essentially the same way as separating conjunction from Separation Logic, except that we give a name $H$ to the conjoined heaps $H_1 * H_2$. We define $\models_{\Sigma_H} s$ as the satisfaction relation such that $\models_{\Sigma_H} φ$ holds if $I(s) = \text{true}$ for all heap formulae $φ$. We also say that $φ$ is valid if $\models_{\Sigma_H} φ$ holds for all $s$, and satisfiable if $\models_{\Sigma_H} φ$ holds for at least one $s$.

2.2 The Assertion Language $\text{CLP}(H)$

We now extend $H$ with user-defined constraints, in the form of recursive definitions. For convenience, we use the framework of Constraint Logic Programming (CLP) [11] to inherit its syntax, semantics, and most importantly, its built-in notions of unfolding rules. To keep this paper self-contained, we now provide a minimal background on CLP.

An atom is of the form $p(\overline{t})$ where $p$ is a user-defined predicate symbol and $\overline{t}$ a tuple of $H$ terms. A rule is of the form $A; -Ψ, B$ where the atom $A$ is the head of the rule, and the sequence of atoms $B$ and the constraint $Ψ$ constitute the body of the rule. The constraint $Ψ$ is a $H$ predicate. 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 substitution $θ$ simultaneously replaces each variable in a term or constraint $e$ into some expression, and we write $eθ$ 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 integer or heap variable into its intended universe of discourse: an integer or an heap. Where $Ψ$ is a constraint, a grounding of $Ψ$ results in true or false in the usual way.

A grounding of an atom $p(\overline{t})$ is an object of the form $p(\overline{t}θ)$ having no variables. A grounding of a goal $G ≡ (p(\overline{t}), Ψ)$ is a grounding of $p(\overline{t})$ where $Ψθ$ is true. We write $[G]θ$ to denote the set of groundings of $G$.

Let $G ≡ (B_1, \ldots , B_n, Ψ)$ and $P$ denote a non-final goal and a set of rules respectively. Let $R ≡ A; -Ψ_1, C_1, \ldots , C_m$, 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

$$(B_1, \ldots , B_{i−1}, C_1, \ldots , C_m, B_{i+1}, \ldots , B_n, B_i = A, Ψ, Ψ_1)$$

provided the constraint $B_i = A ∧ Ψ ∧ Ψ_1$ 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 3 (Unfold). Given a program $P$ and a goal $G$, unfold($G$) is $\{G′|∃R ∈ P : G′ = \text{reduct}(G, R)\}$.
predicates which are true when the rules are read as traditional implications. For example, the following CLP(ℋ) predicate list(l, L) specifies a skeleton list constraint. In other words, the semantics of the rules below dictates that all true groundings of list(l, L) are such that l is an integer, L is a heap which contains a skeleton list starting from l.

\[
\begin{align*}
\text{list}(l, L) &:= l = 0, \ L \equiv \Omega. \\
\text{list}(l, L) &:= l \neq 0, \ L \equiv (l \mapsto n) + L1, \ \text{list}(n, L1).
\end{align*}
\]

We can similarly define predicates for trees and arrays.

**Definition 4** (Assertion Language w.r.t. CLP(ℋ)). Let R denote a CLP(ℋ) program defining one or more predicates \( \Pi \). An atomic assertion is either a Heap predicate, or of the form \( p(t_1, \ldots, t_n) \), \( n \geq 1 \), where \( p_i \in \Pi \). An assertion is simply a boolean combination atomic assertions.

### 2.3 Automated Program Verification

We now adapt the familiar method of Hoare logic to a vanilla imperative language over integer expressions, with the usual if and while control constructs. In addition to integer expressions, we will also consider heap manipulation statements:

- **heap access** \( (x := [y]) \) sets \( x \) to the value pointed to by \( y \);
- **heap assignment** \( ([x] := y) \) sets the value \( x \) points to as \( y \);
- **heap allocation** \( (z := \text{alloc}()) \) sets \( z \) to point to a freshly allocated heap cell.
- **heap free** \( (\text{free}(z)) \) deallocates the cell pointed to by \( z \).

In examples it is convenient to adapt the notation for field selection in records, e.g. \( x \cdot \text{next} \). We assume that such an expression means \([x + \alpha]\) for some (statically defined) offset \( \alpha \). Also note that in the programming language we write \( e \cdot x \) to denote the value at \( x \) in (the global) heap, in the assertion language we may have other heaps, say \( H1 \). We then write \( H1[x] \) to denote the the value at \( x \) in this particular heap. More formally, when the expression \( H1[x] \) containing an occurrence of \( H1[x] \) can be understood as the expression \( e(n) \land H1 \equiv (x \mapsto n) \), where \( e(\cdot) \) is an abbreviation for \( \forall x. (x \mapsto n) \).

In traditional Hoare logic, program verification can be founded on the strongest postcondition axiom

\[
\phi \vdash x := e[\exists x \cdot \psi \cdot \phi / x'[\cdot / x]]
\]

where \( \phi \) is an assertion. Compound usage of this will generate all the required verification conditions for loop free programs. For a loop, the verification condition is simply that the strongest postcondition obtained implies the given loop invariant, and the verification process continues after the loop with the loop invariant and loop termination condition as the current assertion.

We will assume this basic mechanism, which is clearly akin to symbolic execution with assertion checking. That is, in order to prove the Hoare triple \( (\phi) \vdash S(\psi) \) for a loop-free program \( S \), we simply generate strongest postcondition \( \psi' \) along each of its straight-line paths and obtain the verification condition \( \psi' \models \psi \). (The handling of loops can be reduced to this loop-free setting because of user-provided invariants.)

We shall make one further assumption for convenience: that \( S \) is written in static single assignment format where no variable is assigned more than once. This means that the postcondition \( \psi \) will speak of the latest reincarnations of the variables it is interested in.

Now Separation Logic [25] is itself an extension of Hoare Logic. Given the similarity, we re-encode the axioms of Separation Logic in Fig. 4. Note that these axioms manipulate the global heap that is represented by a distinguished heap variable \( H \). Under this treatment, \( H \) is an implicit program variable\(^3\) of type Heap that is assumed to be threaded throughout the program. Thus for example, the formula \( \exists H', \nu : H \equiv (x \mapsto \nu) * H' \) represents that pointer \( x \) is allocated in the global heap \( H \).

Note that while the name \( H \) is the heap variable representing the global heap, other names such as \( x \) and \( y \) represent the program variables. However, in considering loop free programs in which some program variables are changed, we must change the proof obligations accordingly. For example, consider the Hoare triple below which deals with two statements: the first checks that \( z \) is not null, and the second moves \( z \) along.

\[
\{\text{seg}(x, z) \} \ H[z + 1] \neq 0 ; \ z := H[z + 1] \{\text{seg}(x, z)\}
\]

In our setting, we would instead prove:

\[
\{\text{seg}(x, z) \} \ H[z + 1] \neq 0 ; \ z := H[z + 1] \{\text{seg}(x, z)\}
\]

where \( z \) is SSA-introduced, and then it is used appropriately in renaming the postcondition. We finally mention heap assignments. Because we use heap variables explicitly in formulas, this issue gets translated as follows. In the above example, there is no change to the heap. However, consider proving:

\[
\{x \neq 0, [x] = 0 \} \ {x = [x] + 1 \ [x] \geq 1}
\]

Formally, we would propagate the precondition through the program, and obtain:

\[
x \neq 0, n \geq 0, H \equiv (x \mapsto n) \land H3, H2 \equiv (x \mapsto n + 1) \land H3
\]

where \( H \) and \( H3 \) are now existential variables (as is the integer variable \( n \)), and \( H2 \) now represents the (current) global heap. Therefore, what is needed to complete the proof is to show that this formula implies \( H2[x] \geq 1 \) (and not \( H[x] \geq 1 \)).

### 3. The Proof Method

We consider proof obligations of the form \( L \models R \) where \( L \) and \( R \) are CLP goals. This entailment means that \( l m(P) \models \forall x. (L \rightarrow R) \), where \( lm(P) \) denotes the least model of the definite program which defines the recursive predicates — called assertion predicates — occurring in \( L \) and \( R \). \( \forall x. \) is an abbreviation for \( \forall x_1 \ldots \forall x_l (j \geq 0) \) where \( x_i (1 \leq i \leq j) \) is a variable in \( L \).

In other words, the validity of the formula expresses the fact that \( R \) succeeds w.r.t. the CLP(ℋ) program at hand whenever \( L \) succeeds, for any grounding of \( \theta \) of \( L \).

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

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

- First, select an recursive term \( p \in L \), unfold \( L \) wrt. \( p \) and obtain the goals \( L_1, \ldots, L_n \). The validity of the original obligation can now be obtained by ensuring the validity of all the proof obligations \( L_i \models R(1 \leq i \leq n) \).
- Second, select an recursive term \( q \in R \), unfold \( R \) wrt. \( q \) and obtain the goals \( R_1, \ldots, R_m \). The validity of the original obligation can now be obtained by ensuring the validity of any one of the proof obligations \( L \models R_j (1 \leq j \leq m) \).

\(^2\)Here we assume the (de)allocation of single heap cells. This can be generalized.

\(^3\)The variable is “implicit” in the sense that it is not explicitly represented in the syntax of the programming language.
While the obligations are terms, the second element is a recursive term. We will explain this process later in more detail.

Each element in the obligations is a pair, of which the first element is a goal $A$ and the second is a set of premises $B$. The role of proof obligations is to capture the state of a proof. The combination of (CP) and (SUB) rules attempts, what we call, a direct proof. In principle, it is similar to the process of “matching” in (U+M) paradigm. For brevity we then use $A |= B$ to denote the fact that the validity of $A$ is known, and $B$ can be proved directly by using only (SUB) and (CP) rules.

The left unfold with induction hypothesis (LU+1) is the key rule. It selects a recursive term $q$ on the LHS and performs a complete unfold of the LHS wrt. the term $p$, producing a new set of proof obligations.

We now present a formal calculus for the proof of $L \models R$. Other than unfold-and-match, the power of our proof framework comes from the key concept: induction. Of course, the obvious challenge how to automate the use of induction in the proof process.

Definition 5 (Proof Obligation). A proof obligation is of the form $A \models L \mid R$ where the $L$ and $R$ are goals and $A$ is a set of pairs $⟨A; p⟩$, where $A$ is an assumption goal and $p$ a recursive term.

The role of proof obligations is to capture the state of a proof. Each element in $A$ is a pair, of which the first element is a goal $A$ whose truth can be assumed inductively. $A$ acts as a dynamic induction hypothesis and can be used to to transform the proof obligation at hand. We will explain this process later in more details. The second element is a recursive term $p$, to which the application of a left unfold gives rise to the addition of the induction hypothesis $A$.

Our proof rules — the obligation on top, and its reduced form at the bottom — are presented in Figure 5. $A \models B$ denotes that the validity of $A$ is known, and $B$ can be proved simply by consulting an SMT solver. While $\text{gen}(p)$ denotes the time when the recursive term $p$ is generated during the proof process, $\text{kill}(p)$ denotes the time when the recursive term $p$ is unfolded and removed.

Given $L \models R$, our proof shall start with $\emptyset \models L \models R$, 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.

The substitution (SUB) rule removes one occurrence of a assertion predicate, say $p(y)$, 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 terms 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 terms (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 (U+M) paradigm. For brevity we then use $A \models B$ to denote the fact that the validity of $A$ is known, and $B$ can be proved directly by using only (SUB) and (CP) rules.

The left unfold with induction hypothesis (LU+1) is the key rule. It selects a recursive term $q$ on the LHS and performs a complete unfold of the LHS wrt. the term $p$, 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 use of induction later in the proof. For technical reason needed below, we do not just add the obligation $L \models R$ as an assumption, but also need to keep track of the term $p$. This is why in the rule we see a pair $⟨L \models R; p⟩$ added into the current set of assumptions $A$.

On the other hand, the right unfold (RU) rule selects some recursive term $q$ and performs an unfold on the RHS of a proof obligation wrt. $q$. The RU rule does not necessarily obtain all the reducts. In the proof process, the two unfold rules will be systematically interleaved.

Example 1. Consider the following proof obligation (for simplicity, we are not interested in separation):

\[ A \models x \neq 0, \text{list}(x) \models \text{lseg}(x,y), \text{list}(y) \]

where list and lseg are defined as follows:

\[
\begin{align*}
\text{list}(x) & \equiv x = 0 \lor \text{emp} \lor \\
& \text{list}(x) \lor t = x_{\text{next}} \land \text{list}(t) \\
\text{lseg}(x,y) & \equiv x = y \lor \text{emp} \lor \\
& t \neq 0 \land y = t_{\text{next}} \land \text{lseg}(x,t)
\end{align*}
\]

Below we show how this proof obligation can be successfully dispensed by applying of (SUB), (RU), and (CP) rules in sequence.

\footnote{We note that for brevity and ease of understanding, the recursive predicates used in our examples will be defined similarly as in traditional separation logic.}
such a restriction, there is no danger of cyclic reasoning when associated to the predicate being defined so that any recursive “call” n program properties, e.g., require some well-founded metric to which the application of (\(A\theta\)) ensure progressiveness in our proof process. Otherwise, assuming the validity of \(\theta\) to obligations harder than the original one, we require that at least one of the remaining two entailments, namely \(L \models L'\) and \(R \models R'\), can be discharged quickly by a direct proof.

In (1A-1) rule, given the current obligation \(p(\bar{x}) \land L_1 \land L_2 \models R\) and an assumption \(A \equiv p(\bar{y}) \land L' \models R'\), we choose (1) \(p(\bar{x}) \land L' \land L_2\) to be our \(L\) and (2) \(R' \land L_2\) to be our \(R\). We can see that the validity of \(L \models L'\) directly follows from the assumption \(A\theta\). One restriction onto the renaming \(\theta\) to avoid cyclic reasoning, is that \(\theta\) must rename \(\bar{y}\) to \(\bar{x}\) where \(p(\bar{x})\) is a term which has been generated after \(p(\bar{y})\) had been unfolded. Such fact is indicated by \(gen(p(\bar{x})) \geq k(\bar{x}, \bar{y})\) in our rule. Another side condition for this rule is that the validity of \(L \models L'\), or equivalently, \(L_1 \models L'\theta\) can be discharged immediately by a direct proof.

In (1A-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, \(p(\bar{y}) \land L' \land L_2\) serves as our \(L\) while \(R' \land L_2\) serves as our \(R\). The validity of \(L \models L'\) trivially follows from the assumption \(A\theta\), namely \(p(\bar{x}) \land L' \models R' \theta\). As in (1A-1), we also put similar restriction upon the renaming \(\theta\). Another side condition we require is that the validity of \(L \models R\) can be discharged immediately by a direct proof. At this point we could clearly see the duality nature of (1A-1) and (1A-2).

Now let us briefly and intuitively explain the restriction upon the renaming \(\theta\). Here we make sure that \(\theta\) renames term \(p(\bar{y})\) to term \(p(\bar{x})\), 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 our proof process. Otherwise, assuming the truth of \(A\theta\) in constructing the proof for \(A\) would not be valid. This is the reason why for each element of \(A\), we not only keep track of the assumption, say \(L' \models R'\), but also the recursive term \(p\) to which the application of (\(U+1\)) gives rise to the addition of \(L' \models R'\).

We mention here that some works on recursive rules for defining program properties, e.g., require some well-founded metric \(n\) associated to the predicate being defined so that any recursive “call” would have to be associated with a metric \(n\) such that \(m < n\). The non-recursive part of the definition would have measure \(0\). With such a restriction, there is no danger of cyclic reasoning when applied in our inductive framework. This is essentially because an unfolded obligation, when required to behave like an obligation with metric \(n\) before the unfolding, can only be associated to another expression whose metric is less than \(n\). We however consider this kind of restriction on the rules as a significant imposition.

It is important to note that, our framework as it stands, does not require any consideration of a base case, nor a well-founded measure. Instead, we place 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.

**EXAMPLE 2.** Consider the obligation relating two definitions of list segments (again, for simplicity, we are not concerned with separation constraints):

\[ lseg1(x, y) \models lseg(x, y) \]

where \(lseg1\) is defined as below:

\[ lseg1(x, y) \equiv x = y \land \emp \lor x \neq 0 \land t = x \cdot \next \cdot lseg1(t, y) \]

(U+M) will not be able to prove this obligation since \(\text{Unfold}\) and \(\text{lseg1}\) can never be reduced to the same predicate so that matching can take place. However, our framework can discharge this obligation by applying (1A-1) rule twice. For space reason, in Fig. 6, we only show the most interesting path of the proof tree.

**EXAMPLE 3.** Consider the obligation relating the definitions of linked list and list segment (again, for simplicity, we are not concerned with separation constraints):

\[ \text{lseg}(x, y), \text{list}(y) \models \text{list}(x) \]

Figure 7 shows how we can discharge this obligation via the use of (1A-2) rule. Without (1A-2) rule, we cannot automatically prove this obligation using all the remaining rules. Let us pay a close attention at the step where we apply the induction application. For the sake of discussion, assume that instead of (1A-2) we now attempt to apply rule (1A-1). The requirement for \(\theta\) forces it to rename \(x\) to \(y\) and \(y\) to \(t\). However, the side condition \(L_1 \models L' \land y \neq \next\) cannot be fulfilled, since \(t \neq 0, y = t \cdot \next\). \(\text{list}(y) \models \text{list}(t)\).

Now return to the attempt of (1A-2) rule. The RHS of the current obligation matches with the RHS of the only assumption 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 \(lseg(x, t) \equiv lseg(x, y)\theta\). Note that \(lseg(x, t)\) was indeed generated after \(lseg(x, y)\) had been unfolded and removed. The remaining transformation is then straightforward.

We conclude this Section with a theorem stating the conditions in which a proof can be completed.

**THEOREM 1** (Soundness). An obligation \(\L \models R\) holds if, starting with \(\emptyset \vdash L \models R\), there exists a sequence of applications of proof rules that results in an empty set of proof obligations.  

**Proof Sketch.** The soundness of rule (CP) is obvious. The rule (RU) is sound because when \(R' \notin \text{Unfold}_d(R)\) then \(R' \models R\). Therefore, the proof of the obligation \(A \vdash L \models R\) can be replaced by the proof of the obligation \((A \land \text{Unfold}_d(L)) \models L \models R\) since \(L \models R'\) is stronger than \(L \models R\). Similarly, the rule (SUB) is sound because \(L \land p(\bar{x}) \models R\) and \(\text{Unfold}_d(L \land p(\bar{x})) \models \emptyset \models \text{Unfold}_d(L \land p(\bar{y}))\). The rule (\(U+1\)) is partially sound in the sense that when \(\text{Unfold}_d(L) = \{L_1, \ldots, L_n\}\), then proving \(L \models R\) can be substituted by proving \(L_1 \models R, \ldots, L_n \models R\). This is because...
in the least-model semantics of the definitions, \( L \equiv R \)\( C \equiv R \)
\( C \) can soundly transform the current proof obligation \( B \) to the set of assumed assertions \( A \) avoid cyclic reasoning and therefore can safely perform induction on \( A \).

We now describe a systematic algorithm for dispensing a proof obligation \( A \equiv \{ \text{assertions} \} \) and \( B \equiv \{ \text{goals} \} \) in our algorithm are straightforward, except for a few noteworthy

### 4. The Algorithm

We now describe a systematic algorithm for dispensing a proof obligation \( L \models R \) mechanically. The main algorithm is in Figure 9.

In the figure, we use \( X \cup Y \equiv Z \) to denote \( X \supseteq X \cap Y \).

We start off by calling the function \texttt{Prove} with the original obligation \( L \models R \), the set of assumptions \( A \) to be \( \emptyset \), and all the counters \( lb, rb, ib \) to be 0. The counters \( lb, rb, ib \) are to keep track of, respectively, how many left unfolds, right unfolds, and inductions have been applied in this current path. These counters are to ensure that our algorithm terminates.

**Base Case:** The function \texttt{DirectProof} acts as the base case of our recursive algorithm. This function corresponds to the repetitive applications of the (SUB) rule and a successful query to an SMT solver, after treating all recursive predicates in the LHS as uninterpreted (CP)-rule.

Intuitively, \texttt{DirectProof} succeeds if the proof obligation is simple “enough” such that a proof by matching can be achieved. We note here that, our proof rules in Section 3 allow other rules (e.g. (RU) rule in Example 1) to interleave with the (SUB) and (CP) rules. However, in our deterministic algorithm, applications of (SUB) and (CP) rules are coupled together within the function \texttt{DirectProof}.

Let us examine the function \texttt{DirectProof}. If there is a recursive predicate \( q \) on the RHS, but not in the LHS, the function returns immediately, indicating failure with \( \bot \). Otherwise, the function then proceeds by finding some (not exhaustive) substitutions \( \Theta \) such that with each \( \theta \in \Theta \), we can simultaneously remove all the recursive predicates on the RHS. This process will remove most existential variables on the RHS, since existential variables usually appear in some recursive predicates.

In case there remain some existential variables on the RHS, we attempt to bind them with the obvious candidates on the LHS (therefore extend \( \theta \) to \( \theta' \)). After this attempt, if the RHS contains no existential variables, we then call an SMT solver for entailment check. If the answer is yes, \( \theta' \) is returned, indicating that a direct proof has been achieved.

**Recursive Call:** We collect all possible transformations of the current proof obligation, using (IA-1), (IA-2), (LU+1), (RU) rules, into a set of obligations \texttt{OrSet}. The current proof obligation can be successfully discharged if there is any \texttt{Obs} \( \in \texttt{OrSet} \) (\texttt{Obs} itself is a set of proof obligations), where we can discharge every proof obligation \( ob \in \texttt{Obs} \).

The realizations of the proof rules in our algorithm are straightforward, except for a few noteworthy points:

1. Our induction applications will not exhaustively search for all possible candidates. Instead, we only search for some trivial renaming which meet the side conditions of the rules.
2. When we perform left unfold, an obligation which is trivially true, i.e. the non-recursive part of the LHS is unsatisfiable, is immediately removed.
3. If the current obligation contains the LHS and RHS which contradict each other, right unfold will be avoided. The proof for this obligation can succeed only if there is no model satisfies the LHS (so only left unfolds are required).

We proceed our search for proof in a depth first search (DFS) manner. The order in which the sets of obligations \( \texttt{Obs} \in \texttt{OrSet} \) are
function ProveAll(Obs)
    (34) foreach ((L \|= R, \A, lb, rb, ib) \in Obs)
    (35) if (\neg Prove(L \|= R, \A, lb, rb, ib)) return false;
endfunction

function DirectProof(L \|= R)
    (36) if (\exists q(x) \in R such that \exists y(q) \in L) return ⊥
    (37) L' := get_all_recur(L)
    (38) R' := get_all_recur(R)
    (39) Θ := \{ substitution θ \in R' | θ \subseteq L' \}
    (40) if (Θ = \{ \}) return ⊥
    (41) foreach (θ ∈ Θ) continue
    (42) Φ := get_all_nonrecur(L)
    (43) Ψ := get_all_nonrecur(R)
    (44) θ' := bind_remaining_existential_variables(Ψ, Φ, θ)
    (45) if (has_existential_variables(Ψ, Φ, θ')) continue
    (46) if (entailment(Φ, Ψθ')) return θ'
    (47) return ⊥
endfunction

Figure 9: Supporting Functions

5. An obligation resulted from a left unfold will be ordered before those resulted from a right unfold (since it allows induction).

Example 4. Revisit the obligation in Example 1:

\emptyset \vdash x \neq 0, \text{list}(x) \vdash \text{lseg}(x,y), \text{list}(y)

For simplicity we ignore the information about the counters lb, rb, and ib.

First, the call to DirectProof fails since there is the predicate lseg which appears in the RHS but not in the LHS. Inductions do not take place either, as the set of assumptions is currently empty.

We perform left unfold first. There is only one recursive predicate on the LHS. Let \A' be

\{ x \neq 0, \text{list}(x) \vdash \text{lseg}(x,y), \text{list}(y) \}.

The obligation

\A' \vdash x \neq 0, x = 0 \vdash \text{lseg}(x,y), \text{list}(y)

is immediately discharged since the LHS is unsatisfiable. We then end up with a set of one obligation, as the result for our left unfold:

\O_{\emptyset} \equiv \{ \A' \vdash x \neq 0, z = x \cdot \text{next}, \text{list}(z) \vdash \text{lseg}(x,y), \text{list}(y) \}

We proceed with right unfold, producing four sets, each consists of one obligation as follows:

\O_1 \equiv \{ \emptyset \vdash x \neq 0, \text{list}(x) \vdash \text{lseg}(x,y), y = 0 \}
\O_2 \equiv \{ \emptyset \vdash x \neq 0, \text{list}(x) \vdash \text{lseg}(x,y), y \neq 0, z = y \cdot \text{next}, \text{list}(z) \}
\O_3 \equiv \{ \emptyset \vdash x \neq 0, \text{list}(x) \vdash x = y, \text{list}(y) \}
\O_4 \equiv \{ \emptyset \vdash x \neq 0, \text{list}(x) \vdash x = t \neq 0, y = t \cdot \text{next}, \text{lseg}(x,t), \text{list}(y) \}

Assume that the initial order of those singleton sets of obligations are as shown above. After the first two passes, the order between those singleton sets is the same. The third pass, however, moves the singleton set \O_2 to the first position. The fourth pass, on the other hand, moves \O_1 to the second position. The fifth pass keep \O_0 at the third position. The remaining two singleton sets, namely \O_2 and \O_4 are tied and placed at the end.

We also comment that by proceeding with the first singleton obligation set, namely \{ \emptyset \vdash x \neq 0, \text{list}(x) \vdash x = y, \text{list}(y) \},

Figure 8: The Algorithm

considered might heavily affect the efficiency, but not the effectiveness, of our algorithm. Such order is dictated by our heuristics, as the call to function OrderByHeuristics (line 30) indicates.

We now discuss our simple heuristics which governs the implementation of the function OrderByHeuristics. Most of them are intuitive and directly follow from the fact that our base case is only reached by a successful call to DirectProof function.

We proceed by a number of passes. In each pass, we first order the obligations within each Obs ∈ OrderSet. We then consider the order of OrderSet by comparing the last obligation in each set Obs ∈ OrderSet. Subsequent passes will not undo the work of the previous passes, instead will only work on the obligations and/or sets of obligations which are tied in previous passes.

1. An obligation which has contradicting LHS and RHS, given by the function contradict will be ordered after those do not (since the chance to successfully discharge such obligation is small).

2. An obligation contains no recursive predicates on the RHS will be order before those contain some recursive predicate(s) on the RHS.

3. An obligation having a recursive predicate q such that q appears in the RHS but not in the LHS will be ordered after those not.

4. An obligation contains more existential variables which cannot be deterministically bound to some non-existential variables (variables on the LHS) will be ordered after those contains less.

function Prove(L \|= R, \A, lb, rb, ib)
    /* Natural proof, i.e. by unification and smt */
    if (DirectProof(L \|= R)) return true

    let L = Φ, p_1, ..., p_n and R = Ψ, q_1, ..., q_m
    OrSet := ∅
    (2) if (ib < INDUCTIONBOUND) /* Induction Application */
        foreach (p_1(y) \& L \|= R; p_1(y) \in A)
        Find (q_1(y) \in L s.t. gen(p_1(y)) ≥ θ1(p_1(y)))
        if (x = yθ and θ is a valid renaming)
        Lnew:=L \{ p_1(x) \} \union R \θ
        OrSet := ∪ \{ Lnew \|= R, \A, lb, rb, ib + 1 \}
    (3)
    foreach (p_2(y) \& L \|= R; p_2(y) \in A)
    Find a valid renaming θ \union s.t. x = yθ
    OrSet := ∪ \{ L \\{ p_2(x) \} \|= L'θ, \A, lb, rb, ib + 1 \}
    (4)
    foreach (lb < MAXLEFTBOUND) /* Left Unfold */
    if (lb < MAXRIGHTBOUND and ¬ contradict(L \|= R)) /* Right Unfold */
    foreach (p_i \in L)
    Ob := ∅
    (17) foreach (L_j \in \{ L_1, L_2, ..., L_n \}) := UNFOLD(p_j))
    ob := \{ L_j \\union L \\{ p_j \} \|= R, \A', lb + 1, rb, ib)
    (18)
    if (trivially true(\emptyset)) continue
    (19)
    Ob := Ob \union \{ ob \}
    (20)
    if (Ob = ∅) return true else OrSet := \{ Ob \}
    (21)
    foreach (q_i \in R)
    if (lb < MAXRIGHTBOUND and ¬ contradict(L \|= R)) /* Right Unfold */
    foreach (q_i \in R)
    Ob := ∅
    (26)
    if (Ob = ∅) return false
    (27)
    OrSet := OrderByHeuristics(OrSet)
    (28)
    foreach (Ob \in OrSet)
    if (ProveAll(Ob)) return true
    (29)
endfunction
a direct proof of it is successful. Therefore the original obligation is discharged successfully. The corresponding sequence of applications of proof rules is shown below, which is slightly different from what shown in Section 3.

(\text{RU}) \quad \emptyset \vdash x \neq 0, \text{list}(x) \models \text{lsseg}(x, y), \text{list}(y)

(\text{SUB}) \quad \emptyset \vdash x \neq 0, \text{list}(x) \models x = y, \text{list}(y)

(\text{CP}) \quad \emptyset \vdash x \neq 0, \text{list}(x) \models x = x

5. Experimental Evaluation

Our prototype is built upon CLP(R) [12] to control the search and unification process, while making connections to Z3 SMT solver [7]. We evaluated it on a 3.2Gz Intel processor and 2GB RAM machine, running Linux. In our experiments below, the values for $\text{INDUCTIONBOUND}$, $\text{MAXLEFTBOUND}$, $\text{MAXRIGHTBOUND}$ are respectively 3, 5, 5.

Real Programs. We focus our evaluations on open-source library programs, collected and published by [21]. These 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 (as shown in Table 1).

<table>
<thead>
<tr>
<th>Program</th>
<th>Function</th>
<th>Time(s) / Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>glib/glist.c</td>
<td>free, prepend, concat, copy, insert_before, remove_all, remove_link, delete_link, reverse, nth, nth_data, find, position, index, last, length, append, remove, insert_at_pos, insert_sorted_list, insert_sorted_list, merge_sorted_list, merge_sort</td>
<td>$&lt; 1$ s</td>
</tr>
<tr>
<td>Doubly Linked-List</td>
<td>free, prepend, reverse, nth, nth_data, position, find, index, last, length</td>
<td>$&lt; 1$ s</td>
</tr>
<tr>
<td>Queue</td>
<td>simpoleq_init, simpoleq_remove_after, simpoleq_insert_head, simpoleq_insert_tail, simpoleq_insert_after, simpoleq_remove_head</td>
<td>$&lt; 1$ s</td>
</tr>
<tr>
<td>ExpressOS/cachePage.c</td>
<td>lookup_prev, add_cachepage</td>
<td>$&lt; 1$ s</td>
</tr>
<tr>
<td>ExpressOS/memoryRegion.c</td>
<td>memory_region_init, create_user_space_region, split_memory_region</td>
<td>$&lt; 1$ s</td>
</tr>
<tr>
<td>linux/mmap.c</td>
<td>find_vma, remove_vma, remove_vma_list, insert_vma_struct</td>
<td>$&lt; 1$ s</td>
</tr>
</tbody>
</table>

Table 1. Verification of open-source libraries.

For all the data structures in this collection, in order for U+M techniques (e.g., [5, 21]) to work, the use of lemmas/axioms is necessary. As already stated in Section 1, the first reason is that most programs contain iterative loops, where the traversal order of the data structures can be different from what suggested by the recursive definitions (e.g., OpenBSD/queue.h). Besides, axioms/lemmas are also needed when dealing with multiple recursive definitions. One typical use is to make a connection between a list segment and a list. As an example, for append function in glib/glist.c file, the list segment, lseg(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. On the other hand, we have successfully automated the proofs of these benchmark programs, using induction, which in short has leveraged the program verification task to a new level of automation.

Though this benchmark is practically significant, the properties to be proven are limited to the inherent properties of the data structures themselves, e.g., appending two singly-linked lists results in another singly-linked list. In practice, there is a much wider zone where the properties of interest are application-specific, and that the code and data structures serve to realize these properties. For example, in an employee database application, one may prove that a tree depicting the managerial hierarchy is such that a manager’s salary always exceeds his/her subordinates. In such circumstances, we believe our proof technology would demonstrate an even bigger impact.

We finally mention that the running time for each function is always less than 1 second, even with ones that take DRAD much longer time to prove. E.g., consider simpoleq_insert_after function, which is used to insert an element into a queue. This example requires reasoning on different predicates, which DRAD needs the help from an axiom and 18 seconds to prove. In other words, in addition to having a higher level of automation, our framework has a potential advantage of being more efficient than frameworks that use lemmas/axioms. This is especially when such formulas get complicated with large disjunctions\(^5\). This is because such disjunctions will ultimately be passed along to the underlying SMT solver. And though efficient in practice, SMT solvers still face a combinatorial explosion challenge as they dissect the disjunction.

Academic Examples. To further highlight the capability of our proof framework in dealing with a variety of data structures, we tested it with a number of academic programs selected from the benchmarks of [5, 21, 20].

Table 2 summarizes the result of verifying different data-structure algorithms by our prototype. It shows that our prototype not only can work with various data structures such as sorted list, cyclic list, heap, binary search tree (BST), AVL tree, Red-Black tree, but also run very fast (i.e., less than one second).

6. Related Work

There is a vast literature on program verification considering data structures. Most of the algorithms require some manual guidance, and so we will only briefly mention these.

The well known formalism of Separation Logic [25] is often combined with a recursive formulation of data structure properties. Implementations, however, are incomplete [2], or deal only with fragments [1, 18]. There is also literature on decision procedures for restricted heap logics; we mention just a few examples: [9, 22, 23, 14, 24, 4, 3]. 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 [15], VCC [6] and Verifast [10] which require significant ghost annotations, and annotations that explicitly express and manipulate frames. However, they do not directly verify the general and therefore complex properties addressed in this paper, but instead can resort to inter-

---

\(^5\)We believe that the axioms in [21] are unnecessarily too complicated, partly because the authors want to minimize the number of axioms.
Our contribution is mainly based on the automatic use of induction, thus breaking through the U+M barrier. In the literature, there have been works on automatic induction. They are concerned carefully discussed in Sec. 1.\r\n\r\nFurthermore, a base case needs to be proven. We refer to the recent paper [16] for further discussion and bibliography of this subject. In contrast to the literature, our notion of induction hypothesis is completely different. Our hypothesis is discovered dynamically, that is, it can be any formula arising from the proof search process. Importantly, in our induction setting, the notions of well-founded measure and base case are irrelevant.

We finally mention the work [13] which provides the foundation of the method in this paper. It described a general method to prove implications between recursive predicates expressed in the CLP [11] framework, and introduced a novel rule of induction (there called coinduction, since no explicit base case is needed). The novelty was that certain proof obligations could be used as induction hypotheses when the proof progresses. The current paper extends [13] 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. More precisely, this allows us to employ a form of induction coercion: a method of defining a new formula whose proof allows induction to take place. Another technical advance is our introduction of timestamps in the two induction rules as an efficient method to avoid circular reasoning. Finally, the present paper focusing on program verification and uses a specific domain of discourse involving the use of explicit symbolic heaps and separation.

### References


We presented a framework to reason about dynamically manipulated data structures. The main contribution was an algorithm which provided a new level of automation across a wider class of programs. Its key technical feature was an ability to automatically employ induction by a systematic consideration of dynamically generated possibilities as induction hypotheses. Finally, experimental evidence showed that the algorithm meets the automation level of the state-of-the-art algorithms. However, these prior algorithms remain significantly limited, notably by their lack of reasoning between different recursive definitions, which our algorithm handles.

### Table 2. Verification of various data-structure algorithms.

<table>
<thead>
<tr>
<th>Data-structure</th>
<th>Function</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted List</td>
<td>find_rec, insert_rec, delete_all_rec, insert_sort_rec, merge_rec, quick_sort_iter</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>Cyclic List</td>
<td>insert_front, insert_back_rec, delete_front, delete_back_rec</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>Heap</td>
<td>heapify_rec</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>BST</td>
<td>find_rec, insert_rec, delete_rec, find_iter, insert_iter, delete_iter</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>AVL-Tree</td>
<td>leftmost_rec, insert_rec, delete_rec, balance</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>RB-Tree</td>
<td>leftmost_rec, insert_rec, delete_rec</td>
<td>&lt; 1s</td>
</tr>
</tbody>
</table>


