Enhancing Inductive Entailment Proofs in Separation Logic with Lemma Synthesis

Quang Loc Le *

Teesside University, UK

Abstract. This paper presents an approach to lemma synthesis to support advanced inductive entailment procedures based on separation logic. We first propose a mechanism where lemmas are automatically proven and systematically applied. Our lemmas may include universal guard and/or unknown predicate. While the former is critical for expressivity, the latter is essential for supporting relationships between multiple predicates. We further introduce lemma synthesis to support (i) automated inductive reasoning together with frame inference and (ii) theorem exploration. For (i) we automatically discover and prove auxiliary lemmas during an inductive proof; and for (ii) we automatically generate a useful set of lemmas to relate user-defined or system-generated predicates. We have implemented our proposed approach into an existing verification system and tested its capability in inductive reasoning and theorem exploration. The experimental results show that the enhanced system can automatically synthesize useful lemmas to facilitate reasoning on a broad range of non-trivial inductive proofs.

Keywords: Lemma Synthesis, Induction Proving, Theorem Exploration, Separation Logic, User-Defined Predicate.

1 Introduction

Separation logic (SL) [20,36] has been well established for reasoning about heap-based programs. Frame rule in SL enables modular (compositional) reasoning in the presence of the heap and is essential for scalability [22,42,9]. In the last decade, a large number of proof systems for SL have been studied [4,6,9,18,10,27]. Generally speaking, the key challenges of these systems are to support bi-abduction (automated frame inference [4,10] and logical abduction [9,24,41]), and automated induction proving [6] in SL fragments with inductive predicates. While the use of general inductive predicates attains expressive power, a powerful entailment procedure supporting the inductive predicates needs to meet the following two main challenges.

Induction Reasoning. Entailment checks involving inductive predicates normally require induction. For an indirect solution, existing works employed lemmas that are consequences of induction. These lemmas were either hardcoded for a set of predefined predicates (i.e. lists [4]), or automatically generated (for normalization of shape analysis [24], or for some predefined classes of predicates [17]). While these lemma approaches

* This work was done in part when the author was a PhD student in National University of Singapore.
can handle induction reasoning for some specific predicates, they could not support induction in general proofs. Brotherston et al. make an important step towards automating general induction reasoning in SL with cyclic entailment proofs \[6,8\]. Recently, authors in \[12\] managed induction by a framework with historical proofs. However, these proof systems \[6,12\] did not consider frame inference. Thus, they provide limited support for the frame rule as well as modular verification.

**Completeness.** Past works introduce decision procedures for SL decidable fragments including hardwired lists \[4,14,31\], or even user-defined predicates with some syntactic restrictions \[18\]. However, the SL fragment with (arbitrary) user-defined predicates (and arithmetic constraints) is in general, undecidable. Entailment procedures for this general fragment typically trade off completeness for expressiveness \[6,10,16,27\].

To enhance the completeness of program verification, there have been efforts of exploring relations between predicates via *user-supplied* lemmas \[4,6,8,16,22,30\]. While such a static approach puts creative control back into the users’ hands, it is not fully automatic and is infeasible to support inductive proofs relying on auxiliary lemmas of dynamically synthesized predicates (like those in \[7,24\]).

In this work, we propose an approach to lemma synthesis for advanced inductive proofs in a SL fragment with user-defined predicates and Presburger arithmetic. Our technical starting point is an entailment procedure for user-defined predicates (i.e. those procedures in the spirit of \[10\]) combined with second-order bi-abduction \[24\]. We extend this proof system with a new mechanism where lemmas are automatically generated, proven and systematically applied. Finally, we apply lemma synthesis into theorem exploration.

Frame inference has been studied in SL entailment procedures like \[4,10,32\]. Intuitively, these systems prove validity of an entailment \( \Delta_a \vdash \Delta_c \) by unfolding user-defined predicates, subtracting heap predicates until halting at sequents with empty heap (i.e. the emp predicate in SL) in the consequent, such that \( \Delta_f \vdash \text{emp} \land \pi \) and then, conclude the entailment is valid with the frame \( \Delta_f \), denoted by \( \Delta_a \vdash \Delta_c \leadsto \Delta_f \). However, these systems did not provide a direct solution for general induction proofs. We tackle this challenging of frame inference for inductive entailment proofs via the new lemma synthesis. Our key insight is that the entailment check \( \Delta_a \vdash \Delta_c \leadsto \Delta_f \) is semantically equivalent to the check \( \Delta_a \vdash \Delta_c \leadsto \text{emp} \). To infer frame for inductive entailment checks like the former, we will prove the latter check inductively while inferring frame \( \Delta_f \) abductively. Concretely, we assume frame as an unknown predicate \( U \), construct the conjecture lemma \( l_1 : \Delta_a \rightarrow \Delta_c \rightarrow U \), and finally inductively prove this lemma and abductively infer a definition of the predicate \( U \). The benefit of the use of lemma synthesis in our approach is twofold. First, our proposed approach is easily integrated into existing proof systems with lemma mechanism i.e. \[30,16\]. Second, the synthesized lemmas (i.e. \( l_1 \)) are accumulated for reuse in future.

Lemmas in our system may include universally quantified guards and unknown predicates. We will use the notation \( \Delta[\bar{v}] \) to stand for a formula with free variables \( \bar{v} \). Our lemmas with universal guard \( G \) have the form \( \forall \bar{v} \cdot H(\bar{v}) \land G(\bar{v}) \rightarrow B(\bar{v}) \) whereas universal guard \( G \) and body \( B \) may include unknown predicates whose definitions need to be inferred. While guards over universal variables make our mechanism very ex-

---

1 We will refer lemma with universal guard as universal lemma.
pressive, unknown predicates enable us to synthesize generic lemmas i.e. those with weak(est) guards and strong(est) bodies. The meaning of lemmas is interpreted in classic semantics \cite{20} i.e. for the lemma above \( \forall \bar{v} \cdot H(\bar{v}) \land G(\bar{v}) \Rightarrow B(\bar{v}) \); the LHS exactly entails the RHS (with empty heap in residue). Our lemma proving is based on the principle of cyclic proof \cite{6,8}, and can support induction proving. For universal lemma synthesis, given the entailment check \( \exists \bar{e} \cdot \Delta_a \land G(\bar{e}) \vdash \Delta_c \), our system would generate the lemma \textbf{lemma l2}: \( \forall \bar{e} \cdot \Delta_a \land P(\bar{e}) \Rightarrow \Delta_c \), whereas \( P \) is a newly-inferred predicate.

Our predicate inference mechanism is based on the principle of second-order bi-abduction (SOBD), for shape domain \cite{24} and for pure (non-shape) domains \cite{41}, which is an extension of bi-abduction \cite{9} to user-defined predicates. A SOBD entailment procedure takes two SL formulas \( \Delta_{ante} \) and \( \Delta_{conseq} \) as inputs. It infers missing hypotheses \( R \) and residual frame \( \Delta_{frame} \). \( R \) is a set of Horn clause-based constraints over unknown variables of \( \Delta_{ante} \) and \( \Delta_{conseq} \). These constraints have the form of logical implication i.e. \( \Delta_L \Rightarrow \Delta_R \). The set \( R \) can be solved to obtain definitions of the unknown predicates by the algorithms in \cite{24,41}.

In inductive reasoning, theory exploration is a technique for automatically generating and proving useful lemmas for sets of given functions, constants and datatypes \cite{13,23,28}. In our context, theory exploration is meant for discovering useful lemmas for a set of given user-defined predicates. In SL, this technique was indeed presented \cite{41,17}. Like \cite{17} and unlike \cite{4}, our approach is automation-based. Different from these, our technique is capable of generating lemmas with newly-inferred user-defined predicates i.e. those that are synthesized while proving the lemmas.

The novelty of our proof system is the lemma synthesis with predicate inference. Our primary contributions are summarized as follows:

- We propose a new mechanism where universal lemmas are soundly synthesized and systematically applied.
- We synthesize lemmas to support inductive proofs together with frame inference and theorem exploration.
- We have implemented the proposal in an entailment procedure, called \( S \), and integrated \( S \) into a modular verification. Our experiments on sophisticated inductive proofs show that our approach is promising for advancing state-of-the-art in automated verification of heap-manipulating programs. \( S \) and benchmarks are available at \url{http://loris-5.d2.comp.nus.edu.sg/s2}.

2 Motivation and Overview

2.1 Entailment Procedure using Universal Lemma Synthesis

We extend an entailment procedure with basic inference rules (i.e. \cite{6,10}) to inference capability with the second-order bi-abduction (SOBD) mechanism \cite{24} and a new lemma mechanism. This lemma mechanism enhances the inference rules with a set of external and proven lemmas \( L \). These lemmas can be initially supplied by users as well as additionally and dynamically synthesized while the entailment is proven. The enhanced entailment procedure is formalized as follows: \( \Delta_{ante} \vdash_L \Delta_{conseq} \Rightarrow (R, \Delta_{frame}) \) such that: \( L \land R \land \Delta_{ante} \Rightarrow \Delta_{conseq} \land \Delta_{frame} \). Consequently, inference rules of the starting system are using the augmented lemmas \( L \) in the new system as the following:
For a standard proof system for SL with inductive predicates, please refer to [10]. A summary is presented in App. [A] We shall propose \([LAPP]\) and \([LAPP-v]\) rules for lemma application (Sec. [4]), \([LSYN]\) rule for lemma synthesis, \([R_s]\) rule for heap split, and \([AU], [AF], [AU-P], [AF-P]\) rules for predicate inference (Sec. [5]).

To illustrate how our proof system can support induction reasoning together with complex frame inference, consider the following entailment check \(\forall C_0\)

\[
\forall C_0: lll_{\text{last}}(t,t') + lll(y) \vdash t' \rightarrow c_1(q)
\]

pred \(lll_{\text{last}}(\text{root}) \equiv \text{emp} \land \text{root} = \text{null} \lor \exists q \cdot \text{root} \rightarrow c_1(q) \ast lll(q)\)

pred \(lll_{\text{last}}(\text{root}, s) \equiv \text{root} \rightarrow c_1(\text{null}) \land \text{root} = s \lor \exists q \cdot \text{root} \rightarrow c_1(q) \ast lll_{\text{last}}(q, s)\).

whereas \(\text{struct } c_1\{c_1.s = \text{next} ; \} \). \(\forall C_0\) is a verification condition generated to verify memory safety of programs that access the last element (i.e. \(\text{glist_append}\) in glib.c of GLIB library) - see App. [C.2]. In the predicate definitions above, we use basic SL notations to express heaps, e.g. empty predicate (i.e. \(\text{emp}\)), points-to predicate (i.e. \(\text{root} \rightarrow c_1(p)\) asserts a concrete heap cell bound with an allocated data type \(c_1\), pointed-to by the variable \(\text{root}\) and linked with downstream pointer \(p\) via the field \(\text{next}\).)

To infer frame for this inductive entailment check, we assume the frame be an unknown predicate, i.e. \(U_2(t,t', #.q,y)\), and form the following conjecture:

\[
\text{lemma c: } lll_{\text{last}}(t,t') + lll(y) \rightarrow t' \rightarrow c_1(q) \ast U_2(t,t', #.q,y)
\]

Then, we prove its validity and infer a set of relational assumptions as:

\[
\begin{align*}
\sigma_1: & \ast lll(y) \land t' \land q = \text{null} \Rightarrow U_2(t,t',q,y) \\
\sigma_2: & t \rightarrow c_1(q_1) \ast U_2(q_1,t',q_2,y) \Rightarrow U_2(t,t',q_2,y)
\end{align*}
\]

From \(\sigma_1\) and \(\sigma_2\), we synthesize the following definition for \(U_2\)

\[
U_2(\text{root}, t', q, y) \equiv lll(y) \land \text{root} = t' \land q = \text{null} \lor \exists q_1 \cdot \text{root} \rightarrow c_1(q_1) \ast U_2(q_1,t',q,y)
\]

Finally, using theorem exploration presented in Sec [6] (and App. [B]), we generate the following two-way \(\text{separating lemma}\) to normalize the predicate \(U_2\):

\[
\text{lemma conseq } U_2(\text{root}, t', q, y) \leftrightarrow U_3(\text{root}, t') + lll(y) \land q = \text{null}
\]

\[
U_3(\text{root}, t') \equiv \text{emp} \land \text{root} = t' \lor \exists q_1 \cdot \text{root} \rightarrow c_1(q_1) \ast U_3(q_1,t')
\]

To sum up, our system successfully proves and derives \(U_3(t,t') + lll(y) \land q = \text{null}\) as frame of \(\forall C_0\). We present an example for universal lemma synthesis in App. [C.1]

### 2.2 Modular Verification with Lemma Synthesis

```
1 void check(struct c2* a)
2 { while(a->val==1)a=a->next;
3   while(a->val==2)a=a->next;
4   assert a->val==3;
5   return;
}
```

Fig. 1. Code of method check.

We refer \(A \leftrightarrow B\) as two-way lemma, a short form of two reverse lemmas: \(A \rightarrow B\) and \(B \rightarrow A\).
Consider the code fragment in Fig. 1. This code fragment checks whether list segment pointed by a is decomposed into three regions: a list segment of 1 values (the while loop at line 2), a list segment of 2 values (the while loop at line 3), and a 3-value node (the assertion at line 4). We assume that the method check (while loop at line 2, line 3) has been supplied with the conditions 1 (s1, s2, and s3 resp.) specification as:

\[(s_1)\text{ requires } ls1(a,p_1) \lor ls2(p_1,p_2) \land p_2 \rightarrow c_2(3,\text{null})\text{ ensures true;}
\]

\[(s_2)\text{ requires } ls1(a,p_3) \lor p_3 \rightarrow c_2(v_1,p_4) \land v_1 \neq 1\text{ ensures } ls1(a,p_3) \lor p_3 \rightarrow c_2(v_1,p_4) / \land a' = p_3;
\]

\[(s_3)\text{ requires } ls2(a,p_5) \land p_5 \rightarrow c_2(v_2,v_p) \land v_2 \neq 2\text{ ensures } ls2(a,p_5) \land p_5 \rightarrow c_2(v_2,v_p) / \land a' = p_5;
\]

whereas \(a'\) is value of a after the loop; the data structure and the predicates \(ls1, ls2\) are defined as: struct \(c_2\{\text{int val}; \text{struct c}_2\text{ next}\};\)

\[
\begin{align*}
\text{pred } &ls1(root,s) \equiv \text{emp} \land \text{root} = s & \text{pred } &ls2(root,s) \equiv \text{emp} \land \text{root} = s \\
\lor &\exists q \cdot \text{root} \rightarrow c_2(1,q) \land ls1(q,s) & \lor &\exists q \cdot \text{root} \rightarrow c_2(2,q) \land ls2(q,s);
\end{align*}
\]

As a (bottom-up) modular verification, the loops are verified prior to the verification of the method check; and the correctness of a method is reduced to the validity of appropriate verification conditions generated. Our system generates verification conditions to ensure absence of memory errors (no null dereference, no double free and no memory leak), validity of functional calls/loops via compositional pre-/post-conditions and post-conditions holding. For illustrating the proposed approach, we briefly discuss the reasoning on the verification condition (VC1) that was generated before the loop at line 2 (to prove that the current context can imply the loop invariant):

\[ls1(a,p_1) \land ls2(p_1,p_2) \land p_2 \rightarrow c_2(3,\text{null}) \vdash \text{false} \land ls1(a,p_3) \lor p_3 \rightarrow c_2(v_1,p_4) / \land v_1 \neq 1\]

(The lemma store \(L = \emptyset\text{ means there are no user-supplied lemmas.}\) VC1 requires both induction reasoning (for the \(ls1(a,p_1)\text{ predicate) and frame inferences to prove safety of the rest of the program. Hence, this entailment is beyond the capability of most existing SL verification systems (like [30][6][27][32]).

Instead of instantiating and deducing a frame like [4][10], we assume frame as an unknown predicate and infer this predicate via abduction. This inference was implemented in the proposed inference rule \([\text{LSYN}].\) Concretely, the proof of VC1 is as follows:

\[
\begin{align*}
\text{lmsyn}(\Delta_2, \Delta, \emptyset) \rightarrow &\text{(lemma } l_3: \Delta_1 \rightarrow \Delta, \ast U(root,p_1,p_2,p_3,p_4,v_1).\text{true) } \\
ls1(a,p_1) \land ls2(p_1,p_2) \land p_2 \rightarrow c_2(3,\text{null}) \vdash &\Delta_1[\text{a/root}] \rightarrow \text{emp} \land \text{true} \\
ls1(a,p_1) \land ls2(p_1,p_2) \land p_2 \rightarrow c_2(3,\text{null}) \land &\text{false} \land ls1(a,p_3) \lor p_3 \rightarrow c_2(v_1,p_4) / \land v_1 \neq 1 \vdash (\text{true} \land U(\bar{v}))
\end{align*}
\]

whereas \(U(\bar{w}) \equiv U(a_+,p_1,p_2,p_3,p_4,v_1).\) The lemma \(l_3\) was synthesized as

\[
\begin{align*}
l_1: &ls1(root,p_1) \land ls2(p_1,p_2) \land p_2 \rightarrow \text{c}_2(3,\text{null}) \\
&\rightarrow ls1(root,p_3) \land p_3 \rightarrow \text{c}_2(v_1,p_4) / \land \text{false} \land U(a,p_1,p_2,p_3,p_4,v_1) / \land v_1 \neq 1
\end{align*}
\]

\[
\begin{align*}
pred U(a,p_1,p_2,p_3,v_1) &\equiv \text{root} = \text{null} \land a = p_1 \land p_2 = p_3 \land v_1 = 3 \\
\lor \text{root} \rightarrow c_2(v', v_4) / U(a,p_1,p_2,p_3,v_4) / \land a = p_1 \land a = p_3 \land v_1 = 2 \land v'_1 \neq 1
\end{align*}
\]

The proof derived for the lemma \(l_3\) will be presented in detail in Sec.5.
For automation, we integrated the proposed proof system into S2 [24], a specification inference system. Simultaneously, we enhanced S2 beyond the shape domain. We show how the proposed lemma synthesis was integrated into a modular verification with incremental specification inference over shape and size properties in App. C.3.

3 Preliminaries

A Fragment of Separation Logic. The syntax of the fragment is as follows.

\[ \Phi ::= \Delta \mid \Phi_1 \lor \Phi_2 \quad \Delta ::= \exists \bar{v} \cdot (\kappa \land \pi) \quad \kappa ::= \text{emp} \mid r \rightarrow c(t) \mid P(t) \mid \kappa_1 \land \kappa_2 \]

\[ \pi ::= \pi \land \phi \mid \phi \mid p(t) \quad \phi ::= \alpha \mid \exists v \cdot \phi \mid \forall v \cdot \phi \mid \neg \phi \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \]

\[ \text{pred } P(\bar{v}) \equiv P(\bar{v}) \quad c \in \text{Data Types} \quad t_i, v, r \in \text{Var} \quad t \equiv t_1, \ldots, t_n \]

A formula \( \Phi \) can be a disjunctive formula (\( \Phi_1 \lor \Phi_2 \)). A conjunctive formula \( \Delta \) is conjoined by spatial formula \( \kappa \) and pure formula \( \pi \). All free variables are implicitly universally quantified at the outermost level. \text{null} is a special heap location. In the predicate \( r \rightarrow c(t) \), \( r \) is a root variable. For abduction reasoning, our fragment also includes unknown predicates, spatial (\( \text{U}(t) \)) and pure (\( p(t) \)) second-order variables, whose definitions need to be inferred [24,11]. Pure formulas are constraints over (in)equality \( \alpha \) (on pointers), and Presburger arithmetic \( i \). Note that \( v_1 \neq v_2 \) and \( v \neq \text{null} \) are short forms for \( \neg (v_1 = v_2) \) and \( \neg (v = \text{null}) \), respectively. We occasionally use a sequence \( (i, \bar{t}) \) to denote a set when it is not ambiguous. We omit \( \pi \) when it is \text{true}. A formula without any user-defined predicate instances is referred as a base formula.

User-Defined (\( \text{UD} \)) Predicate. A \( \text{UD} \) predicate \( P \) is defined as \( \text{pred } P(\bar{v}) \equiv \bigwedge_{i=1}^n (\exists \bar{w}_i \cdot \Delta_i) \); whereas \( P \) is predicate name. \( \bar{v} \) is a set of formal parameters. \( \bigwedge_{i=1}^n (\exists \bar{w}_i \cdot \Delta_i) \) is a predicate definition. \( \exists \bar{w}_i \cdot \Delta_i \) (\( i \in 1 \ldots n \)) is a branch of the disjunction. In each branch, we require that variables which are not in formal parameters must be existentially quantified.

Definition 1 (Root Parameter) Given \( \text{UD} \) predicate \( P : \text{pred } P(\bar{v}) \equiv \bigwedge_{i=1}^n (\exists \bar{w}_i \cdot \Delta_i) \); a parameter \( r \in \bar{v} \) is a root if for any base formula \( \kappa \land \pi \) derived by unfolding the predicate instance \( P(\bar{v}) \), \( r \) is either a root of a points-to predicate (\( r \rightarrow \_ \) occurs in \( \kappa \)) or \( r = \text{null} \).

We syntactically detect root parameters as follows. A formal parameter \( r \in \bar{v} \) is a root if any branch \( \kappa_i \land \pi_i \), \( i \in \{1 \ldots n\} \), one of the following four conditions holds: (i) \( r \) is a root variable of a points-to predicate (\( r \rightarrow \_ \) occurs in \( \kappa_i \)); (ii) \( r \) equals to \text{null}; \( \pi_i \implies r = \text{null} \); (iii) \( r \) equals to another root parameter: \( \pi_i \implies r = s \), where \( s \in \bar{v} \); or (iv) \( r \) is a root parameter of another \( \text{UD} \) predicate. Without loss of generality, we will write \( P(r, \bar{v}) \) to indicate that \( r \) is a root parameter. A predicate with multiple root parameters will be considered to transform into multiple predicates with single root parameter in Sec. C.6 Unfolding. The function unfolding(\( \Delta, j \)) unfolds the jth \( \text{UD} \) predicate instance, i.e. \( P^j(\bar{v}) \), of the formula \( \Delta \). The steps are formalized as follows:

\[
\text{unfolding}(\exists \bar{w}_0 \cdot P^j(\bar{v}) \cdot \kappa_0 \land \pi_0, j) \sim \bigwedge_{i=1}^n (\exists \bar{w}_i \cdot \Delta_i \cdot \kappa''_i \land \pi''_i) \]

First, the function looks up the definition of \( P \), refreshes the existential quantifiers. Second, formal parameters are substituted by the actual parameters. Finally, substituted
definition is combined (and normalized) with residual formula as in the RHS of \(\sim\). We will refer to recursive predicate instances of unfolded heap formula \(\kappa_i\) as descendant predicate instances (of the \(j^{th}\) \(\mathsf{UD}\) predicate instance).

4 Lemma Mechanism

Lemma Formalism. In general, lemmas in our system are formalized as:

\[
\text{lemma id: } \forall \nu \exists \bar{u}_1 \bar{R}_i \Delta_1(\nu) \rightarrow \exists \bar{u}_2 \bar{R}_r \Delta_r(\nu).
\]

Our lemmas are used to relate two reachable heap regions starting from a same root pointer of the root predicates \(\bar{R}_i\) and \(\bar{R}_r\). A root predicate is a points-to or a \(\mathsf{UD}\) predicate. Without loss of generality, we require that the root pointer of a root predicate is explicitly denoted with the preserved name \(\text{root}\). For frame inference, LHS and RHS must capture the same heaps (i.e. \(\forall \nu \exists \bar{u}_1 \bar{R}_i \Delta_1(\nu) \vdash_L \exists \bar{u}_2 \bar{R}_r \Delta_r(\nu) \sim (\_ \text{emp} \wedge \pi)\). We occasionally use \(\text{id}\) to indicate the lemma.

Lemma Application. During proof search, proven lemmas are considered as external inference rules. We assume that free variables of a lemma are renamed to avoid clashing before the lemma is applied. The application of a lemma \(\text{id}\) is formalized as follows.

\[
\begin{align*}
\frac{\text{Lemma Application}}{\text{LAPP}}
\end{align*}
\]

The lemma \(\text{id}\) is applied into the above entailment through three steps. First, we match the predicate \(\bar{R}_i\) of the antecedent with root predicate \(\bar{R}_r\). (We note that \(\bar{R}_r\) must contain a root pointer.) \(\bar{R}_i\) and \(\bar{R}_r\) are matched and unified by a partial function \(\text{match}\). If the matching is successful, \(\text{match}\) produces substitutions as follows:

\[
\text{match}(\text{root} \rightarrow c(\bar{f})), x \rightarrow c(\bar{u}) = [\bar{u}/\bar{f}] \circ [x/\text{root}]
\]

\[
\text{match}(P(\text{root}, \bar{f}), P(x, \bar{w})) = [\bar{w}/\bar{f}] \circ [x/\text{root}]
\]

Second, we prove guard and identify \(\text{cut}\) of the antecedent in the first line. Last, we combine the residue of the antecedent with RHS of the lemma before continuously proving the consequent (of the entailment) in the second line.

Universal Lemma Application. The universal guard \(\forall G\) is equivalent to infinite conjunction \(\bigwedge_{\rho} G[\rho]\). \(\forall G\) makes the lemmas with universal guard more expressive. However, it also prevents applying these lemmas into entailments with instantiated (existential) guards as the second step of the above lemma application rule could not be established (i.e. \(\exists x \Delta \not\vdash \forall x \Delta\)). While mechanism in [30] was based on “delayed guard”, we now propose guard instantiation, a sound solution for universal lemma application. In order to apply a universal lemma into an entailment with instantiated (existential) guard, we present a technique that dynamically and intelligently instantiates universal guard (of the lemma) before really deploying this lemma. Concretely, our technique will substitute universal guard by a finite set of its instances. The soundness of this technique is based on the following theorem.

Theorem 1 (Lemma Instantiation). Let \(\Delta_{\text{ante}}\) and \(\Delta_{\text{conseq}}\) be antecedent and consequent of an entailment check. Let \(L\) be a set of universal lemmas and let \(L'\) be a
set of instantiated lemmas obtained by substituting each $\forall \bar{v}G(\bar{v})$ in a universal guard by a finite conjunction of its instantiations. If $\Delta_{ante} \vdash L' \Delta_{conseq}$ is valid then so is $\Delta_{ante} \vdash L \Delta_{conseq}$.

Proof. Semantically, a guard with universal variables $\forall \bar{v}G(\bar{v})$ implies a conjunction of its instantiations. As such universal guard $\forall \bar{v}G(\bar{v})$ is on the left-hand side of lemmas, a proof system with inference rules $L'$ implies the corresponding proof system with inference rules $L$. Thus, if $\Delta_{ante} \vdash L' \Delta_{conseq}$ is valid then so is $\Delta_{ante} \vdash L \Delta_{conseq}$. □

Inspired by quantifier instantiation in Simplify [15], we symbolically select “relevant” instantiations over universal variables of a guard by looking up substitutions over the universal variables such that the instantiated lemma suffices to prove the validity of a given entailment. The substitutions are selected through shape predicate matching [10] (the corresponding action used in [16] is subtracting.). The application of lemmas with universal variables is formalized as:

$$\begin{align*}
\text{(lemma id } & \forall \bar{v} \cdot \text{R}_1 \Delta_1 \land G(\bar{v}) \rightarrow \Delta_r) \in L \land \rho = p_m \circ \text{ins}(\bar{v}) \land \Delta_1 \vdash L \Delta_2 \rightarrow \rho \rightarrow (\Delta_1, R_{\bar{r}_1}) \\
\rho_m = & \text{match}(R_1, R_1) \rightarrow \Phi_{\bar{r}_1} \Delta_1 \rho \rightarrow L \Delta_2 \rightarrow \rho \rightarrow (\Delta_2, \Phi_{\bar{r}_2}) \\
& \rightarrow (\Delta_3, \Phi_{\bar{r}_3}) \rightarrow \rightarrow (\Delta_3 \land R_3, \Phi_{R})
\end{align*}$$

In the rule above, after head matching, we rename universal variables and obtain the substitution by the function $\text{ins}$. This substitution helps to instantiate quantified variables in both universal guard of the lemma and existential guard of the antecedent.

Symbolic “relevant” instantiations are selected during the course of the left entailment at line 2 and are captured in the residue $\Phi_{\bar{r}_2}$. Finally, “relevant” instantiations are used to prove the instantiated guard of the right entailment at line 2. We implement the selection of the symbolic “relevant” instantiations via by instantiation mechanism in [10]. We summarize the mechanism in App. A

5 Automated Inductive Entailment Procedure

In Fig. 5 we propose new six inference rules: $[\text{LAPP}]$ is for cutting heaps; $[\text{LSYN}]$ is for dynamically generating and proving auxiliary lemmas; and the rest is for generating relational assumptions while checking entailment. $[\text{L}]$ rule, a revision (with frame inference) of the Monotonicity rule [20], helps to generate smaller sub-goals, and then to generate more reusable lemmas. In this rule, $FV$ returns free variables of a formula. Two necessary conditions to apply the rule $[\text{LSYN}]$ are: (i) role predicates of the antecedent (LHS) and the consequent (RHS), $R_0$ and $R_c$, must share a same root pointer; (ii) $R_r$ must be a defined predicate instance i.e. $P(x, \bar{w})$. Here, $\forall\exists(\bar{w}, \pi)$ is an auxiliary function that existentially quantifies in $\pi$ all free variables that are not in the set $\bar{w}$. We present the $\text{lem synth}$ procedure in the subsection 5.1.

In $[\text{AU}]$ and $[\text{AF}]$ rules, $R(\bar{r}, \bar{t})$ is either $\rightarrow c(\bar{t})$ or known (defined) $P(\bar{r}, \bar{t})$, or unknown predicate $U'(\bar{r}, \bar{t}, \bar{w})$. We use $\#$ notation in unknown predicates to guide abduction and proof search. We only abduce on pointers without $\#$-annotated. $U_l(\bar{w}, \bar{t}')$ is another unknown predicate generated to capture downstream heaps. After abduced pointers will be annotated with $\#$ to avoid double abduction. New unknown predicate $U_l$ is only generated if at least one parameter is not annotated with $\#$ (i.e. $\bar{w} \cup \bar{t}' \neq \emptyset$). To avoid
Fig. 2. Inference Rules for Lemma Synthesis with Predicative Inference.

conflict between abduction rules and other (unfolding, subtraction) during proof search, all root pointers in a heap formula must be annotated with # in unknown predicates. For examples, in our system while the formula \( x \rightarrow c_1(y) \cup U_1(x, y) \) is valid, the formula \( x \rightarrow c_2(y) \cup U_1(x, y) \) is invalid. For the check \( x \rightarrow c_1(\text{null}) \cup x \rightarrow c_1(y) \cup U_1(x, y) \), our proof search will apply subtraction the heap pointed by \( x \) rather than abduction. We illustrate the \([\text{AF}]\) rule with the following example:

\[
\begin{align*}
\text{AF} & \quad y \rightarrow c_1(\text{null}) \cup U_2(y, x, y) \vdash \sigma_1 \quad \text{n-\#-annotated} \\
& \quad \text{AF} \quad x \rightarrow c_1(y) \wedge y \rightarrow c_1(\text{null}) \cup U_2(x, x, y) \vdash \sigma_1 \quad \text{AF-\#-P} \\
& \quad \text{AF} \quad x \rightarrow c_1(y) \wedge y \rightarrow c_1(\text{null}) \cup U_2(x, x, y) \vdash \sigma_1 \quad \text{AF-\#-P} \\
& \quad \text{AF} \quad x \rightarrow c_1(y) \wedge U_2(x, x, y) \vdash \sigma_1 \quad \text{AF-\#-P} \\
& \quad \text{AF} \quad x \rightarrow c_1(y) \wedge U_2(x, x, y) \vdash \sigma_1 \quad \text{AF-\#-P} \\
& \quad \text{AF} \quad x \rightarrow c_1(y) \wedge U_2(x, x, y) \vdash \sigma_1 \quad \text{AF-\#-P} \\
& \quad \text{AF} \quad x \rightarrow c_1(y) \wedge U_2(x, x, y) \vdash \sigma_1 \\
\end{align*}
\]

In the second application of \([\text{AF}]\) rule, no new unknown predicate was introduced as there was no pointers without #-annotated. \([\text{AU-\#-P}]\) and \([\text{AF-\#-P}]\) rules abduse on pure and applied on consequent which has empty heap. \(\text{XPure}\) procedure soundly transforms a heap formula into a pure formula \([\text{[10]}]\).

5.1 Lemma Synthesis - \texttt{lemsyn} Procedure

Given the entailment check \( \forall \bar{v} \forall \bar{e} \forall \bar{r} \cup \bar{\pi} \mathcal{L} \rightarrow \mathcal{L} \rightarrow \mathcal{L} \rightarrow \Delta_c \), our \texttt{lemsyn} procedure synthesizes the lemma \( \forall \bar{v} \forall \bar{e} \forall \bar{r} \cup \bar{\pi} \mathcal{L} \rightarrow \mathcal{L} \rightarrow \mathcal{L} \rightarrow \Delta_c \) through three steps: conjecture construction, lemma proving and predicate synthesis.

Conjecture Construction. At this step, \texttt{lemsyn} enriches the original check with either pure unknown predicate \( P(\bar{v}, \bar{e}) \) (for universal lemma) or shape unknown predicate \( U(\bar{v}) \)
(for frame inference). While the former is only added if there exist existential variables in LHS, the latter is only added if the sets of universal variables over pointers of LHS and RHS are not identical. We note that in the former, to prepare for universal guard inference, we need to existentially quantifies the pure formula; in the latter, $\Delta_r$ is a minimum closure of connected heaps of $R_c$. The unknown predicate is generated with parameters that are union of free variables of LHS and RHS. Among these, pointer-based parameters are annotated with $#$ following the principle that instantiation (and subtraction) are done before abduction. The detail is as follows: (i) all intersection variables of LHS and RHS are $#$-annotated; (ii) roots pointers of RHS are $#$-annotated; (iii) remaining pointers are not $#$-annotated.

**Lemma Proving.** Our lemma proving, `lemprove` procedure, is based on the principle of cyclic proof [6,8]. Two steps of this proof technique are: back-link form (i.e. linking current sequents to a historical sequent); and global trace condition checking. We implement these steps via lemma application. The procedure `lemprove` is formalized as:

$$
\forall_{i=1}^{n} \Delta_i \equiv \text{unfold}(\Delta_i, j) \quad \frac{\Delta_i \vdash_{L \cup \{\text{link}\}} \Delta_r \sim (R_i, \text{emp} \land \pi_i)}{(\text{lemma link: } \Delta_l \rightarrow \Delta_r, L) \sim \bigwedge_{i=1}^{n} R_i}
$$

To prove the conjecture `link`, `lemprove` looks up a $j^{th}$ UD predicate instance in $\Delta_l$ to apply $\text{LU}$ rule. For a successful proving, any disjunct $\Delta_i$ obtained from the unfolding of $\Delta_l$ must imply $\Delta_r$ with empty heap in the residue. This unfolded predicate instance is a progressing point. Induction hypothesis is encoded by the lemma `link`: induction on such predicate instance is performed as an application on this lemma into a descendant predicate instance of the unfolded predicate instance. We denote such application is cyclic lemma application. We note that the lemma synthesis may be nested; it means our system would speculate additional lemmas while proving a lemma.

In cyclic term [6,8], the initial lemma is a `bud`, entailment check which applied cyclic lemma application is companion of the `bud` above, and proof that are removed all cyclic lemma applications is a pre-proof. A path in a pre-proof is a sequence of sequent occurrences (entailment checks) derived by applying inference rules. For soundness, a pre-proof must be a cyclic proof; it must satisfy the global trace condition i.e. for every infinite path there is infinitely many progressing points. We state the condition that a proof derived by our system is indeed a cyclic proof in the following lemma.

**Lemma 1 (Soundness).** If all descendant predicate instances of the unfolded $j^{th}$ UD predicate instance have involved in a cyclic lemma application, then $\bigwedge_{i=1}^{n} R_i \land \Delta_l \models \Delta_r$.

**Proof.** A path in the proof is infinite if it includes the descendant predicate instances (of the $j^{th}$ UD predicate instances) as these predicate instances may unfolded infinitely. If every the descendant predicate instances has involved in a cyclic lemma application, it has infinitely progressing points and thus the global trace condition is satisfied. Then our proof is a cyclic proof.

**Predicate Synthesis.** If the conjecture `link` contains unknown predicates, we infer by SOBD a conjunction set of relational constraints $R = \bigwedge_{i=1}^{n} R_i$ (over the unknown predicates) such that $R \land \Delta_l \models \Delta_r$. After that we deploy those algorithms in [24,33]
For each lemma synthesized, we always consider to generate its corresponding relational assumptions are canceled, i.e. not forwarded to outer scope.

Two-way Lemmas. For each lemma synthesized, we always consider to generate its reverse lemma. For each pair of such two-way lemmas, one with unknown predicates will be inferred (and proven); another is substituted with the newly-inferred predicates prior to proven. For the former, we choose the conjecture with more case splits i.e. more \( \forall \) predicates in the LHS. We hope that proving such conjecture will generate more sub-goals and thus more relational assumptions would be generated. The more assumptions our system generates, the more meaningful predicate definitions is synthesized.

5.2 Motivating Example Revisit

Our system started proving \( \text{l_1} \) by unfolding predicate \( \text{l_1}(a,p_1) \) (like \([11]\)) as follows.

\[
\text{l_2}(a,p_2) \rightarrow \text{c_2}(3, \text{null})
\]

\[
\overset{\text{l_1}(a,p_1) \rightarrow \text{c_2}(v_1,p_4) \rightarrow \text{U}(v_1)}{\text{l_2}(a,p_2) \rightarrow \text{c_2}(3, \text{null})}
\]

whereas \( \Delta_3 = 1 \equiv \{a#,p_1,p_2,p_3#,p_4,v_1\} \), \( \bar{v}_3 \equiv \{a#,a,p_2,p_3#,p_4,v_1\} \). (In all tree derivations below, we discard the \( \text{XPURGE} \) rule on top for simplicity.)

**Base Case.** Proof of the left subgoal was derived as:

\[
\text{emp} \land \text{true} \vdash (\text{l_1}; \text{l_3}) \text{emp} \land \text{true} \sim (\text{true}; \text{emp} \land \text{true})
\]

\[
\text{l_2}(a,p_2) \rightarrow \text{c_2}(3, \text{null}) \vdash (\text{l_3}) \text{a} \rightarrow \text{c_2}(v_1,p_4) \rightarrow \text{U}(v_1)
\]

whereas \( \bar{v}_3 \equiv \{a#,a,p_2,a#,p_4,v_1\} \), and the nested conjecture \( \text{l_4} \) was constructed as:

\[
\text{l_4} : \text{l_2}(a,p_2) \rightarrow \text{c_2}(3, \text{null}) \rightarrow \text{a} \rightarrow \text{c_2}(v_1,p_4) \rightarrow \text{U}(v_1)
\]

Proof of the conjecture \( \text{l_4} \) was derived as in Fig. 3 whereas \( \tilde{v}_3 \equiv \{a#,a,a,a#,p_4,v_1\} \).
(\bar{v}_1) \equiv (a#.,a,p_2,a,.p'_3,v_1'), (\bar{v}_5) \equiv (a#.,a,p_2,a,.p'_4,v_1,p_4'). Inferred assumptions are:

\sigma_1: p_4 = \text{null} \land v_1 = 3 \Rightarrow U(a,a,a,a,p_4,v_1)
\sigma_2: p_4 \rightarrow \bar{c}_2(v'_1,p'_3) \land U(a,a,a,p_2,a,p_4,v_1,p'_3) \land v'_1 \neq 1 \land v_1 = 2 \Rightarrow U(a,a,a,p_2,a,p_4,v_1)
\sigma_3: U(a,a,p_2,a,.p'_4,v_1') \land v'_1 \neq 1 \land v_1 = 2 \Rightarrow U_1(a,a,a,p_2,a,p_4,v_1,p'_4,v'_1)

Since \psi_1 was introduced at this scope, before returning to the outer scope, \psi_1 has been synthesized as \text{pred}_1(a,a,p_2,a,.p_4,v_1,.root,v'_1') \equiv \exists p_4 U(a,a,p_2,a,.root,v'_1') \land v'_1' \neq 1 \land v_1 = 2. The set assumptions forwarded to the outer scope is: \mathcal{R}_1 \equiv \sigma_1 \land \sigma_2.

**Induction Case.** Proof of the right subgoal was derived as:

\[ \begin{array}{c}
\text{M} \quad \text{LAPP} \quad \text{AF} \\
\text{ls1}(a_1,p_5') \ast p_3 \rightarrow \bar{c}_2(v'_1,p'_3) \ast U(v_5) \land v'_1 \neq 1 \Rightarrow \text{ls1}(a_1,p_3) \ast p_3 \rightarrow \bar{c}_2(v_1,p_4) \ast U(\bar{v}) \land v_1 \neq 1 \\
\text{ls1}(a_1,p_3) \ast \text{ls2}(p_1,p_2) \ast \bar{c}_2(3, \text{null}) \Rightarrow \text{ls1}(a_1,p_3) \ast p_3 \rightarrow \bar{c}_2(v_1,p_4) \ast U(\bar{v}) \land v'_1 \neq 1 \\
\text{ls1}(a_1,p_3) \ast \text{ls2}(p_1,p_2) \ast p_2 \rightarrow \bar{c}_2(3, \text{null}) \\
\text{ls1}(a_1,p_3) \ast p_3 \rightarrow \bar{c}_2(v_1,p_4) \ast U(\bar{v}) \land v_1 \neq 1 \\
\text{ls1}(a_1,p_3) \ast \text{ls2}(p_1,p_2) \ast p_2 \rightarrow \bar{c}_2(3, \text{null}) \Rightarrow \text{ls1}(a_1,p_3) \ast p_3 \rightarrow \bar{c}_2(v_1,p_4) \ast U(\bar{v}) \land v'_1 \neq 1 \\
\Rightarrow (\bar{v}_7) \equiv (a_1#.,p_1,p_3,#.p'_3,v'_1') \\
\Rightarrow (\bar{v}_7) \equiv (a_1#.,p_1,p_3,#.p'_4,v_1') \\
\Rightarrow (\bar{v}_7) \equiv (a_1#.,p_1,p_3,#.p'_4,v_1') \\
\Rightarrow U(a_1,p_1,p_3,p_4,v_1) \land v_1 \neq 1 \Rightarrow U(a_1,p_1,p_3,p_4,v_1)
\end{array} \]

Now, the predicate \psi_1 is synthesized from the set of assumptions \sigma_1 \land \sigma_2 \land \sigma_4.

**5.3 Soundness**

**Lemma 2.** The rules \([\text{LSYN}]\) and \([\text{LAPP}]\) in Fig. 5 preserve soundness.

Our \([\text{LSYN}]\) rule is derived from the Monotonicity rule showing that spatial conjunction is monotone with respect to implication [20]:

\[ \Delta_a \models \Delta_{c_1} \quad \Delta_a \models \Delta_{c_2} \]
\[ \Delta_a \models \Delta_{c_1} \ast \Delta_{c_2} \models \Delta_{c_1} \ast \Delta_{c_2} \]

The soundness of \([\text{LSYN}]\) is derived from the meaning of frame, i.e. \Delta_a \models \Delta_{c_r} \models \Delta_{\text{frame}} holds if \Delta_a \models \Delta_{c_r} \models \Delta_{\text{frame}}, and the design of our lemma mechanism, i.e. \text{lemma 1:} \Delta_l \rightarrow \Delta_r

is valid iff \Delta_l \models \Delta_{c_r} \models \text{emp}

 Soundness of abduction rules. Since we only apply proven lemmas, it is sound to assume that lemma store is empty (no user-supplied lemmas) and discard this lemma store in our soundness proofs. We introduce the notation \mathcal{R}(\Gamma) to denote a set of predicate definitions \Gamma = \{U_1(\bar{v}_1) \equiv \Phi_1, \ldots, U_n(\bar{v}_n) \equiv \Phi_n\} satisfying the set of assumptions \mathcal{R}. That is, for all assumptions \Delta_l \rightarrow \Delta_r \in \mathcal{R}, (i) \Gamma contains a predicate definition for each unknown predicate appearing in \Delta_l and \Delta_r; (ii) by interpreting all unknown predicates according to \Gamma, then it is provable that \Delta_l implies \Delta_r, written as \Gamma : \Delta_l \rightarrow \Delta_r.

**Lemma 3.** Given the entailment judgement \Delta_a \models \Delta_{c_r} \models (\mathcal{R}, \Delta_f), if there exists \Gamma such that \mathcal{R}(\Gamma), then the entailment \Gamma : \Delta_a \rightarrow \Delta_{c} \models \Delta_f holds.

Abduction soundness requires that if all the relational assumptions generated are satisfiable, then the entailment is valid.
6 Theorem Exploration

We present a mechanism to explore relations for \(\mathbb{U}\mathbb{D}\) predicates by using lemma synthesis. The mechanism is applied for some sets of either statically user-supplied or dynamically analyser-synthesized predicates. The key idea is that instead of requiring designers of proof systems to write lemmas for a specific \(\mathbb{U}\mathbb{D}\) predicate (like in \[4\]), we provide for them a mechanism to design lemmas for a general class of \(\mathbb{U}\mathbb{D}\) predicates. And based on the design, our system will automatically generate specific and on-demand lemmas for a specific \(\mathbb{U}\mathbb{D}\) predicate. In this section, we demonstrate this mechanism through three such classes. Concretely, our system processes each \(\mathbb{U}\mathbb{D}\) predicate in four steps. First, it syntactically classifies the predicate into a predefined class. Second, it follows structure of the class to generate heap-only conjectures (with quantifiers). Third, it enriches the heap-only conjectures with unknown predicates for expressive constraint inference. Last, it invokes the \texttt{lemprove} procedure to prove these conjectures, infer definitions for the unknown predicates and synthesize the lemmas. Especially, whenever a universal lemma \(L \rightarrow R\) is proven, its reverse (the lemma \(R \rightarrow L\)) is also examined. Proven lemmas will be applied to enhance upcoming inductive proofs.

In the next subsection and App. B we present theorem exploration in three classes of \(\mathbb{U}\mathbb{D}\) predicates. In each class, we present step 1 and step 2; steps 3 and 4 are identical to the \texttt{lemsyn} procedure in Sec. 5.1.

6.1 Generating Equivalence Lemmas

**Step 1.** Intuitively, given a set \(S\) of \(\mathbb{U}\mathbb{D}\) predicates and another \(\mathbb{U}\mathbb{D}\) predicate \(P\) (which is not in \(S\)), we look up all predicates in \(S\) which are equivalent to \(P\). This exploration is applied to any new (either supplied or synthesized) \(\mathbb{U}\mathbb{D}\) predicate. For specification inference, we eagerly substitute a newly-inferred predicate in specifications by its equivalent-matching predicate from the library. This makes inferred specifications more understandable. Furthermore this also helps to avoid induction proving on proof obligations generated from these specifications.

**Step 2.** Heap-only conjecture to explore equivalent relation of two predicates (e.g. \(P(x, \bar{v})\) and \(Q(x, \bar{w})\)) is generated as: \(\exists \bar{t}_1 \cdot P(\text{root}, \bar{v}) \rightarrow \exists \bar{t}_2 \cdot Q(\text{root}, \bar{w})\), whereas \(\bar{t}_1 = \bar{v} \setminus \bar{w}\) and \(\bar{t}_2 = \bar{w} \setminus \bar{v}\). The shared root parameter \(x\) has been identified by examining all permutations of root parameters of the two predicates. For example, with l1n and lsegm in Sec. 2 our system examines conjecture: \texttt{lemma eq1} \(\exists p \cdot l\text{segn}(\text{root}, p, m) \rightarrow l\text{ln}(\text{root}, n)\). At step 3, the unknown predicate \(U\) is added to infer constraints over leaf variables \(p, m, \) and \(n\) as: \(\texttt{lemma eq1} \exists p \cdot l\text{segn}(\text{root}, p, m) \land U_1(p, m, n) \rightarrow l\text{ln}(\text{root}, n)\). At step 4, the conjecture \texttt{eq1} is proven and a definition of \(U_1\) is inferred as: \(U_1(p, m, n) \equiv p = \text{null} \land m = n\). For the equivalence, our system also generates and proves the reverse lemma of \texttt{eq1} as: \(\texttt{lemma eq2} \ l\text{ln}(\text{root}, n) \rightarrow \exists p \cdot l\text{segn}(\text{root}, p, m) \land p = \text{null} \land m = n\).

This technique can be applied to match a newly-inferred definition synthesized by shape analyses (i.e. \[7,24\]) with existing predicates of a supplied library of predefined predicates. For specification inference, we eagerly substitute a newly-inferred predicate in specifications by its equivalent-matching predicate from the library. This makes inferred specifications more understandable. Furthermore this also helps to avoid induction proving on proof obligations generated from these specifications.

7 Implementation and Experiments

We have implemented the proposed ideas into an entailment procedure, called \(\mathcal{S}\), starting from SLEEK entailment procedure \[10\]. \(\mathcal{S}\) invokes Z3 \[26\] to discharge satisfiabil-

\(^3\) \(\mathcal{S}\) was initially implemented for the SLCOMP competition \[?\].
Proven | Cyclic$\text{SL}$ | S | #syn
--- | --- | --- | ---
1 | \(lseg(x,t) \land lseg(t,\text{null}) \vdash_{0} lseg(x,\text{null})\) | 0.03 | 0.06 | 1
2 | \(lseg(x,t) \land t \rightarrow c_{1}(y) \land lseg(y,\text{null}) \vdash_{0} lseg(x,\text{null})\) | 0.03 | 0.08 | 1
3 | \(lseg(x,t) \land lseg(t,\text{null}) \land y \rightarrow c_{1}(\text{null}) \vdash_{0} lseg(x,\text{null})\) | 0.03 | 0.11 | 2
4 | \(lseg(x,t) \land lseg(t,y) \land y \neq \text{null} \vdash_{0} lseg(x,y) \land lseg(y,t)\) | TO | 0.21 | 2
5 | \(lseg(x,t) \land lseg(t,y) \land y \neq \text{null} \vdash_{0} lseg(x,y) \land lseg(y,z)\) | 3.00 | 0.57 | 1

| Ent. | Cyclic$\text{SL}$ | S | #syn |
|---|---|---|---|
| 6 | \(x \rightarrow c_{1}(y) \land r\text{lseg}(y,z) \vdash_{0} r\text{lseg}(x,z)\) | 0.02 | 0.10 | 1
| 7 | \(n\text{lseg}(x,z) \land z \rightarrow c_{1}(y) \vdash_{0} n\text{lseg}(x,y)\) | 0.02 | 0.04 | 1
| 8 | \(n\text{lseg}(x,z) \land n\text{lseg}(z,y) \vdash_{0} n\text{lseg}(x,y)\) | 0.03 | 0.06 | 1
| 9 | \(g\text{lseg}(x,z) \land z \rightarrow c_{1}(y) \vdash_{0} g\text{lseg}(x,y)\) | 0.02 | 0.04 | 1
| 10 | \(g\text{lseg}(x,z) \land g\text{lseg}(z,y) \vdash_{0} g\text{lseg}(x,y)\) | 0.02 | 0.04 | 1
| 11 | \(d\text{lseg}(u,v,x,y) \vdash_{0} g\text{lseg}_{2}(u,v)\) | 0.07 | 0.04 | 1
| 12 | \(d\text{lseg}(w,v,x,z) \land d\text{lseg}(u,w,z,y) \vdash_{0} d\text{lseg}(u,v,x,y)\) | 0.04 | 0.11 | 1
| 13 | \(\text{lsto}(x,z) \land \text{lsto}(x,\text{null}) \vdash_{0} \text{lsto}(x,\text{null})\) | 0.06 | 0.06 | 2
| 14 | \(\text{lsto}(x,z) \land \text{lsto}(x,\text{null}) \vdash_{0} \text{lsto}(x,\text{null})\) | 0.18 | 0.12 | 3
| 15 | \(\text{lsto}(x,z) \land \text{lsto}(z,y) \vdash_{0} \text{lsto}(x,y)\) | 4.33 | 0.88 | 3
| 16 | \(\text{binPath}(x,z) \land \text{binPath}(z,y) \vdash_{0} \text{binPath}(x,y)\) | 0.03 | 0.06 | 1
| 17 | \(\text{binPath}(x,y) \vdash_{0} \text{binTreeSeg}(x,y)\) | 0.12 | 0.08 | 1
| 18 | \(\text{binTreeSeg}(x,z) \land \text{binTreeSeg}(z,y) \vdash_{0} \text{binTreeSeg}(x,y)\) | 0.20 | 0.66 | 1
| 19 | \(\text{binTreeSeg}(x,y) \land \text{binTree}(y) \vdash_{0} \text{binTree}(x)\) | 0.06 | 0.03 | 1
| 20 | \(\text{sortll}(x,\text{min}) \vdash_{0} \text{ll}(x)\) | X | 0.05 | 1
| 21 | \(\text{sortll}(x,\text{min, size}) \vdash_{0} \text{lln}(x,\text{size})\) | X | 0.12 | 1
| 22 | \(\text{sortll}(x,\text{min, size}) \vdash_{0} \text{sortll}(x,\text{min})\) | X | 0.08 | 1
| 23 | \(\text{lsn}(x,y,z_{1}) \land \text{lsn}(x,z_{2}) \vdash_{0} \text{lsn}(x,z_{1}+z_{2})\) | X | 0.12 | 1
| 24 | \(\text{lsn}(x,y,z_{1}) \land \text{lsn}(x,z_{2}) \vdash_{0} \text{lsn}(x,z_{1}+z_{2})\) | X | 0.10 | 1
| 25 | \(\text{lsn}(x,t) \land t \rightarrow c_{1}(y) \land \text{lsn}(y,t) \vdash_{0} \text{lsn}(x,t)\) | X | 0.10 | 1
| 26 | \(\text{avl}(x,\text{size, height, bal}) \vdash_{0} \text{btn}(x,\text{size})\) | X | 0.08 | 1
| 27 | \(\text{tll}(x,\text{ll, fr, size}) \vdash_{0} \text{btn}(x,\text{size})\) | X | 0.07 | 1

Table 1. Entailment Checking with Auxiliary Lemma Synthesis

We have also integrated S into S2 [24] and extended the new system to support a modular verification with partially supplied specification and incremental specification inference. In the following, we experiment S and the enhanced S2 in entailment and program verification problems. The experiments were performed on a machine with the Intel i7-960 (3.2GHz) processor and 16GB of RAM.

**Entailment Check.** In Table 1 we evaluate Cyclic$_{\text{SL}}$ and S on inductive entailment problems without user-supplied lemmas (i.e. $L=\emptyset$). Ent 1-19 are shape-only problems; they were taken from Smallfoot [4] (Ent 1-5), and Cyclic$_{\text{SL}}$ [68] (Ent 6-19). Ent 20-27 are shape-numerical problems. We used sortll for a sorted list with smallest value min, and tll for a binary tree whose nodes point to their parents and leaves are linked as a singly-linked list [18][24]. Time is in second. TO (X) denotes timeout (30s) (not-yet-support, resp.). The last column #syn shows the number of lemmas our generated to prove the entailment check. The experimental results show that S can handle a wider range of inductive entailment problems. The experiments also demonstrate the efficiency of our implementation; for 18 problems which both tools successfully verified, while it took Cyclic$_{\text{SL}}$ 8.29 seconds, it took S only 3.14 seconds.

**Modular Verification for Memory Safety.** We enhance S2 to automatically verify a wide range of programs with a higher level of correctness and scalability. In more detail, it automatically verifies those programs in [50] (e.g. bubble_sort, append method of...
list with tail-like predicate in App. C.2 without any user-supplied lemma. By generating consequence parallel separating lemmas, it also successfully infers shape specifications of methods which manipulate the last element of a singly-linked list (i.e. \texttt{gslist.concat} in \texttt{gslist.c}) and a doubly-linked list (i.e. \texttt{g_list.append} in \texttt{glist.c}) of GLIB library \cite{1} (See App. C.2). By generating equivalence lemmas, matching a newly-inferred \texttt{UD} predicate with predefined predicates in S2 is now extended beyond shape-only domain.

We evaluated the enhanced S2 on the cross-platform C library Glib open source \cite{1}. We experimented on heap-manipulating files, i.e. singly-/doubly-linked lists (\texttt{gslist.c/glist.c}), balanced binary trees (\texttt{gtree.c}) and N-ary trees (\texttt{gnode.c}). In Fig.4 we list for each file the number of lines of code (excluding comments) LOC, number of procedures (while/for loops) \#Pr (\#Lo). \#\sqrt{\text{sec}} and sec. show the number of procedures/loops and time (in second) for which the enhanced S2 can verify memory safety without (wo.) and with (w.) the lemma synthesis component. With the lemma synthesis, the number of procedures/loops was successfully verified increases from 168 (81\%) to 182 (88\%) with the overhead of 0.38 seconds.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|l|l|l|l|}
\hline
File & LOC & \#Pr & \#Lo & wo. \#\sqrt{\text{sec}} & w. \#\sqrt{\text{sec}} & \\
\hline
\texttt{gslist.c} & 698 & 34 & 18 & 41 & 2.19 & 47 \ 2.30 \\
\texttt{glist.c} & 784 & 32 & 19 & 39 & 3.20 & 46 \ 3.39 \\
\texttt{gtree.c} & 1204 & 32 & 8 & 36 & 3.46 & 36 \ 3.48 \\
\texttt{gnode.c} & 1128 & 38 & 27 & 52 & 7.52 & 53 \ 7.58 \\
\hline
\end{tabular}
\caption{Experiments on Glib Programs}
\end{table}

\section{Related Work and Conclusion}

\subsection{Entailment Procedure in SL}
Past works in SL mainly focus on developing decision procedures for a decidable fragment combining linked lists (and trees) with only equality and inequality constraints \cite{4,14,31,32,29}. Smallfoot \cite{34}, provided strong semantic foundations and proof system with frame inference capability for the above fragment. Some optimization on segment feature for the fragment with linked list was presented in \cite{14,31,32,29}. Recently, Iosif et. al. extended decidable fragment to restricted \texttt{UD} predicates \cite{18}. \cite{39} presented a comprehensive summary on computational complexity of deciding entailment in SL with \texttt{UD} predicates. Our work, like \cite{10,34}, targets on an undecidable SL fragment including (arbitrary) \texttt{UD} predicates and numerical constraints. Like \cite{10,34}, we trade completeness for expressiveness. Beyond the focus of \cite{10,34}, we provide inductive reasoning in SL using lemma synthesis.

\subsection{Lemma Mechanism in SL}
Lemma is widely used to enhance the reasoning of heap-manipulating programs. For examples, lemmas are used as alternative unfoldings beyond predicates’ definitions \cite{30,8}, external inference rules \cite{16}, or intelligent generalization to support inductive reasoning \cite{6}. Unfortunately, these systems require user to supply those additional lemmas that might be needed for a proof. In our work, we propose to automatically generate lemmas either dynamically for inductive reasoning or statically for theorem exploration.

\subsection{Induction Reasoning}
For a manual and indirect solution for inductive reasoning in SL, Smallfoot \cite{4} presented subtraction rules that are consequent from a set of lemmas of lists and trees. Brotherston et. al. proposed a \textit{top-down} approach to automate inductive proofs using cycle proof \cite{5}. To avoid infinite circular proof search, the cyclic
technique stops expanding whenever current sequent is a repetition of a similar proof pattern detected from historical proof tree. Cyclic proof was successfully implemented in first-order logic \cite{3}, and separation logic \cite{6}. Circularity rule, a similar mechanism to cyclic, was also introduced in matching logic \cite{37}. \cite{12} managed induction by a frame-work with historical proofs. Our proposal extends these systems with frame inference and gives better support for modular verification of heap-manipulating programs.

**Auxiliary Lemma Generation.** In inductive theorem reasoning, auxiliary lemmas are generated (and proven) either *top-down* to support inductive proofs (e.g. IsaPlanner \cite{19}, Zeno \cite{38} and extension of CVC4 \cite{35}) or *bottom-up* to discover theorem (e.g. IsaCosy \cite{23} and HipSpec \cite{13} and \cite{28}). The center of these techniques are heuristics to generate useful lemmas for sets of given functions, constants and datatypes. Typically, while the top-down proposals (i.e. \cite{38}) suggest new lemmas by replacing some common sub-term in a stuck goal by a variable, the bottom-up proposals (i.e. \cite{13}) generate lemmas to compute equivalence functions for functional programs. In our work, we introduce both top-down and bottom-up approaches into an entailment procedure in SL. To support inductive entailment proofs dynamically, we generate auxiliary conjectures with unknown predicates to infer either universal guard or frame. To support theorem discovery, we synthesize equivalence, split/join/reverse and separating conjectures. This mechanism can be extended to other heuristics to enhance proofs of a widen class of \textit{UD} predicates.

9 Conclusion

Lemmas have been widely used to enhance the capability of program verification systems. However, existing reasoning systems of heap-manipulating programs via separation logic rely on user to supply additional lemmas that might be needed for a proof. In this paper, we have presented a mechanism for applying, proving and synthesizing lemma in a SL entailment procedure. We have shown an implementation that has a higher level of automation and completeness for benchmarks taken from inductive theorem proving and software verification sources. Our evaluation indicates that inductive proofs benefit from both bottom-up and top-down lemmas generated by our new approach. It also shows that synthesized lemmas are relevant and helpful to proving a conjecture. Future work includes extending the incremental inference mechanism to other pure domains, e.g. bag/set domain.

References

1. Glib-2.38.2. https://developer.gnome.org/glib/, 2013. [Online; accessed 30-May-2015].
2. Aws Albargouthi, Josh Berdine, Byron Cook, and Zachary Kincaid. Spatial interpolants. In *ESOP*, pages 634–660. 2015.
3. J. Berdine, C. Calcagno, and P. W. O’Hearn. A Decidable Fragment of Separation Logic. In *FSTTCS ’04*. Springer-Verlag, December 2004.
4. J. Berdine, C. Calcagno, and P. W. O’Hearn. Symbolic Execution with Separation Logic. In *APLAS*, volume 3780, pages 52–68, November 2005.
5. J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *Proceedings of TABLEAUX-14*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.
6. J. Brotherston, D. Distefano, and R. L. Petersen. Automated cyclic entailment proofs in separation logic. In CADE, pages 131–146, 2011.
7. J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. In SAS ’14, number 8723 in LNCS, pages 68–84, 2014.
8. J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In Proceedings of APLAS-10, LNCS. Springer, 2012.
9. Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In POPL, pages 289–300, 2009.
10. W.N. Chin, C. David, H.H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. SCP, 77(9):1006–1036, 2012.
11. Duc-Hiep Chu, Joxan Jaffar, and Quang Loc Le. Personal correspondence. 7 June, 2015.
12. Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. Automatic induction proofs of data-structures in imperative programs. PLDI’15, 2015.
13. Koen Claessen, Moa Johansson, Dan Rosn, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In CADE ’24, volume 7898, pages 392–406, 2013.
14. B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In CONCUR, volume 6901, pages 235–249, 2011.
15. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. J. ACM, 52(3):365–473, May 2005.
16. Dino Distefano and Matthew J. Parkinson J. jstar: Towards practical verification for java. In OOPSLA ’08, pages 213–226, New York, NY, USA, 2008. ACM.
17. Constantin Enea, Mihaela Sighireanu, and Zhilin Wu. On automated lemma generation for separation logic with inductive definitions. ATVA, 2015. To appear.
18. R. Jozif, A. Rogalewicz, and J. Simáček. The tree width of separation logic with recursive definitions. In CADE, pages 21–38, 2013.
19. Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. Journal of Automated Reasoning, 16(1-2):79–111, 1996.
20. S. Ishtiaq and P.W. O’Hearn. BI as an assertion language for mutable data structures. In ACM POPL, pages 14–26, London, January 2001.
21. Shachar Izhaky, Nikolaj Bjørner, Thomas Reps, Mooly Sagiv, and Aditya Thakur. Property-directed shape analysis. In CAV, pages 35–51, 2014.
22. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In NFM, pages 41–55, 2011.
23. Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. J. Autom. Reason., 47(3):251–289, October 2011.
24. Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In CAV, volume 8559, pages 52–68, 2014.
25. Quang Loc Le, H.Long Pham, and Wei-Ngan Chin. Satisfiability solver for separation logic with invariant generation. In Technical Report, SoC, NUS, October 2014. avail. http://loris-7.ddns.comp.nus.edu.sg/ project/s2/s2sat-tr.pdf.
26. Leonardo M. and Nikolaj B. Z3: An Efficient SMT Solver. In TACAS, 2008.
27. Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive proofs for inductive tree data-structures. In POPL, pages 123–136, New York, NY, USA, 2012. ACM.
28. Roy McCasland, Alan Bundy, and Autexier Serge. Automated discovery of inductive theorems.
29. JuanAntonio Navarro Prez and Andrey Rybalchenko. Separation logic modulo theories. In APLAS, volume 8301, pages 90–106, 2013.
30. H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In CAV, pages 355–369, 2008.
31. Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In PLDI, pages 556–566, 2011.
32. R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In Natasha Sharygina and Helmut Veith, editors, CAV, volume 8044, pages 773–789. 2013.
33. C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In ASIAN, pages 331–345, 2006.
34. X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In PLDI, pages 231–242, New York, NY, USA, 2013. ACM.
35. Andrew Reynolds and Viktor Kuncak. Induction for smt solvers. In VMCAI, volume 8931, pages 80–98. Springer Berlin Heidelberg, 2015.
36. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In IEEE LICS, pages 55–74, 2002.
37. Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In OOPSLA ’12, pages 555–574, New York, NY, USA, 2012. ACM.
38. William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An automated prover for properties of recursive data structures. In TACAS, pages 407–421, 2012.
39. Antonopoulos Timos, Gorogiannis Nikos, Haase Christoph, Kanovich Max, and Ouaknine Joël. Foundations for decision problems in separation logic with general inductive predicates. In FoSSaCS 2014, pages 411–425, 2014.
40. Antoine Touhans, Bor-Yuh Evan Chang, and Xavier Rival. Reduced product combination of abstract domains for shapes. In VMCAI, pages 375–395, 2013.
41. M.-T. Trinh, Q.-L. Le, C. David, and W.-N. Chin. Bi-abduction with pure properties for specification inference. In APLAS, pages 107–123, 2013.
42. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O’Hearn. Scalable shape analysis for systems code. In CAV, pages 385–398, 2008.
A Separation Entailment Procedure

\[ \begin{align*}
\text{INC1} & \quad x \rightarrow c(\overline{v}) \ast \Delta_1 \models_L \Delta_2 \land x = \text{null} \Rightarrow (\emptyset, \kappa) \\
\text{INC2} & \quad \Delta_1 \land \pi_{eq} = \text{freeEQ}(\rho) \\
\beta & \quad \pi_{eq} = \text{freeEQ}(\rho) \\
\Delta_1 \land \pi_{eq} \models_L \Delta_2 \land \overline{v} = \overline{w} \Rightarrow (\mathcal{R}, \Phi_f) \\
x \rightarrow c(\overline{v}) \ast \Delta_1 \models_L x \rightarrow c(\overline{w}) \ast \Delta_2 \Rightarrow (\mathcal{R}, \Phi_f) \\
\end{align*} \]

\[ \begin{align*}
\text{REDUCE} & \quad \Delta_1 \land \pi_{eq} \models L \Delta_2 \land \overline{v} = \overline{w} \Rightarrow (\mathcal{R}, \Phi_f) \\
\end{align*} \]

Fig. 5. Basic Inference Rules for Entailment Checking

Entailment procedure between \( \Delta_a \) and \( \Delta_c \) is formalized as follows:

\[ \Delta_a \models_L \Delta_c \Rightarrow (\mathcal{R}, \Phi_f) \]

The entailment outputs residual frame \( \Phi_f \) and a set of relational assumptions \( \mathcal{R} \). (For simplicity, we discard footprints and existential quantifiers of consequent in this discussion.) Inference rules are presented in Fig. 5.

To derive a proof for an entailment check, our system deduces antecedent into two parts (i) relevant one would be subsumed by models of the consequent; (ii) the rest will be inferred as residual frame. To do that, it subtracts (match) heap two sides until heap in the consequent is empty (via the [\text{M}], [\text{LU}], [\text{RU}] inference rules). After that, it semantically check the validity for the implication of the pure part by using external SMT solvers and theorem provers (via [\text{XPURE}] inference rule). Typically, an entailment check is performed as follows.

- **Subtracting.** Match up identified heap chains. Starting from identified root pointers, the procedure keeps matching all their reachable heaps with [\text{M}] and [\text{PRED-M}] rules. The former (latter) rule matches two points-to (user-defined, resp.) predicates in antecedent and consequent if they have an identified root. After that, it unifies corresponding fields of matched roots by using auxiliary function freeEQ(\rho):

  \[ \text{freeEQ}(\{u_i/v_i\}_{i=1}^n) = \bigwedge_{i=1}^n \{u_i = v_i\} \]

- **Unfolding.** Derive alternative heap chains. When the procedure is unable to make a progress on matching, it will look up alternative chains for matching through unfolding heap predicates. While the unfolding in the antecedent ([\text{LU}]) rule does cases split, the unfolding in the consequent ([\text{RU}] rule) does proof search.

- **XPure Reducing.** Reduce entailment checking on separation logic to implication checking on the first order-logic with [\text{XPURE}] rule. This reduction was presented in [25]. When the consequent remains empty heap, e.g. emp \land \pi_c, the procedure
employs \[\text{XPure}\] inference rule to decide the entailment result. Firstly, this rules make use of the \[\text{XPure}\] reduction to transform the combination of remain heaps in the antecedent and footprints into the first order-logic formula on the combination of pure domains, e.g. \(\pi_a\). Then it checks the implication \(\pi_a \implies \pi_c\). Technically, to perform such implication checking, the following satisfiability check is performed: 
\[
sat(\pi_a \land \neg(\pi_c)).
\]
If it returns \text{unsat}, the result of the implication is invalid; it returns \text{sat}, the result of the implication is valid; otherwise, the result of the implication is unknown.

During the heap chains matching, aliasing relation on pointers are considered to introduce alternative proofs via \[\text{ALIAS}\] rule.

**Instantiation Mechanism.** A variable is instantiable if it is an actual parameter of a \(\text{UD}\) predicate instance in the consequent (RHS) and is quantifier-free. This mechanism is applied for predicate matching rule \[10\] (corresponding rule used in \[16\] is subtracting) and predicate folding rule \[10\] (corresponding rule used in \[6\] is unfolding predicate in RHS). Whenever a match of a \(\text{UD}\) predicate instance occurs, the entailment procedure binds its instantiable parameters coming from the RHS with corresponding variables from the antecedent (LHS) and moves the equality constraints to the LHS. Whenever a \(\text{UD}\) predicate instance in the RHS is unfolded, our proof system moves pure constraints over instantiable (actual) parameters of the unfolded formulas to the LHS. Moreover, this mechanism is proven sound and is able to enhance the completeness of entailment procedure for a SL fragment including \(\text{UD}\) predicates with pure properties \[10\].

## B Theorem Exploration

### B.1 Generating Reverse/Split/Join Lemmas

**Step 1.** This subsection explores theorem over segment predicates as follows.

**Definition 2 (Segment Predicate)** A predicate \(SP(r, \vec{v}, s)\) is a segment predicate if \(r\) is a root parameter and for any base formula \(\Delta\) which is derived by unfolding \(SP(r, \vec{v}, s)\), \(s\) is a leaf pointer reached from \(r\). We will refer \(s\) as a segment parameter.

For instance, linked-list segment predicate with size property is defined as follows:

\[
\text{pred glseg}n(root, s, n) \equiv \text{emp} \land \text{root} = s \land n = 0 \lor \exists q \cdot \text{root} \overset{c_1(q)}{\rightarrow} \text{glseg}n(q, s, n-1);
\]

The predicate \text{glseg}n above may be an acyclic list, or a complete cyclic list, or a lasso (an acyclic fragment followed by a cycle). The *acyclic* list segment predicate \(\text{ls}egn\) (Sec. 2) is a special segment predicate. Tree segment predicates can be found in \[6\].

A \(\text{UD}\) predicate is syntactically classified as segment predicate if it has one root parameter \(r\) and one segment parameter \(s\) such that \(s\) is a leaf pointer which is reached from \(r\). A \(\text{UD}\) predicate is syntactically classified as *acyclic* segment predicate if it is a segment predicate and the formula \(r \neq s\) occurs in all inductive branches.

**Step 2.** Reverse lemmas explore relations between reverse directions of linked heaps i.e. forwardly and backwardly linked list from root parameter to segment parameter in inductive branches of segment predicates. Our implementation for reverse lemmas currently restricts for reachable heaps linked by points-to predicates and segment predicate
instances. With a segment predicate $Q(root, \bar{w}, s)$, for each inductive branch $\exists \bar{w}_i \Delta_i$, reverse linked heaps $\exists \bar{w}_i \Delta_i$ of is examined as follows: (i) mark reachable heaps, a set of points-to predicates and segment predicate instances from root to $s$; (ii) swap root and $s$. Now, $s$ is a root variable of either a points-to predicate or a segment predicate instance; (iii) starting from the heap predicate with $s$, reverse the reachable heaps following the links. For each points-to predicate, swap points-to variable with downstream field variable of the links. For each segment predicate instance, swap root parameter and segment parameter; (iv) keep the rest of $\Delta_i$ unchanged.

Reverse conjectures are initially generated over reachable heaps as: $\exists \bar{w}_i \Delta_i \rightarrow Q(root, \bar{w}, s)$.

For example, with segment list $\text{glsegn}$ above, we generate the following lemma: $\text{lemma rev}_1 \exists q \cdot q \rightarrow c_1(s) \land \text{glsegn}(root, q, n-1) \rightarrow \text{glsegn}(root, n, s)$. Heap-only conjecture to explore join relation for the segment predicate $P(x, \bar{w}, s)$ ($x$ is a root parameter and $s$ is a segment parameter) is generated as:

$$\exists z, \bar{w}_1, \bar{w}_2 \cdot P(x, \bar{w}_1, z) \land P(z, \bar{w}_2, s) \rightarrow \exists \bar{w} \cdot P(x, \bar{w}, s)$$

Two heap-only conjectures to explore join relation for the acyclic segment predicate $Q(x, \bar{w}, s)$ of data type $\text{c}(f_1,v_1)$ are generated as:

$$\exists z, \bar{w}_1, \bar{w}_2 \cdot Q(x, \bar{w}_1, \bar{z}) \land Q(z, \bar{w}_2, s) \land \text{null} \rightarrow \exists \bar{w} \cdot Q(x, \bar{w}, \text{null})$$

and

$$\exists z, \bar{w}_1, \bar{w}_2 \cdot Q(x, \bar{w}_1, \bar{z}) \land Q(z, \bar{w}_2, s) \land s \rightarrow c(\bar{z}) \rightarrow \exists \bar{w}, \bar{w}_3 \cdot Q(x, \bar{w}, s) \land s \rightarrow c(\bar{w}_3)$$

Similarly, split heap-only conjecture $\Delta \rightarrow \Delta_1 \land \Delta_2$ is generated as a opposite form of the corresponding join heap-only conjecture $\Delta_1 \land \Delta_2 \rightarrow \Delta$.

### B.2 Generating Separating Lemmas

**Step 1.** This subsection explores relations over $\cup d$ predicates including either parallel or consequence separating parameters. Two parameters of a predicate are parallel separating if they are both root parameters (e.g. those of the predicate $\text{zip}$, Sec. 5.1). Two parameters of a predicate are consequence separating if one is root parameter and another parameter is internal variable reachable from the root in all base formulas derived by unfolding the predicate (e.g. those of the predicate $\text{upost}$, Sec. 5.3). We generate these separating lemmas to explicate separation globally. As a result, the separation of actual parameters is visible from analyses. This visible separation enables strong updates in modular heap analysis or frame inference in modular verification.

**Step 2.** Suppose $r_1, r_2$ are consequence or parallel parameters in $Q(r_1, r_2, \bar{w})$, heap conjecture is generated as: $Q(r_1, r_2, \bar{w}) \rightarrow Q_1(r_1) \land Q_2(r_2) \land \text{null}$. For example, the $\text{zip}$ predicate is suggested to split through the following parallel separating conjecture: $\text{lemma para}_{\text{zip}}(\text{root}, r_2) \rightarrow Q_1(\text{root}) \land Q_2(r_2)$.

### C More Examples

#### C.1 Universal Lemma Synthesis

To illustrate the lemma synthesis, consider the following entailment check $E_i$
\( \exists k. \text{lln}(x, n) \land n \geq k \land k \geq 0 \land i = k \land j = n - k \vdash \exists p. \text{lseg}n(x, p, i) \land \text{lln}(p, j) \)

We define the \( \text{lln} \) (\( \text{lseg}n \)) predicate to describe an acyclic singly-linked list null-terminated (segment, respectively) over the data type \( c_1 \) with size property \( n \) as follows:

\[
\begin{align*}
\text{pred} \text{lln}([n, 0]) & \equiv \text{emp} \\
\text{pred} \text{lln}([n, 0]) & \equiv \text{emp} \\
\forall i, j, n. \text{lln}(i, j, n) & \iff \text{lln}(i, j, n) \\
\end{align*}
\]

whereas \( \text{struct} \{ c_1 \} \) has more predicates, it may need more case splits. Thus we choose \( \text{jn} \) for inference since it would generate more relational constraints and our system can obtain more precise definition of \( P \) predicate.

Inspired by cyclic proof systems [5,6,37,12], our system has employed the lemma \( \text{jn} \) as induction hypothesis for proving \( \mathcal{E}_2 \) and \( \mathcal{E}_3 \). For \( \mathcal{E}_2 \), we subtract (match) the predicate \( \text{lln} \) pointed by \( \text{root} \) in both sides, instantiate \( n \) and generate the following assumption to successfully prove the rest of RHS:

\[ R_2: a = 0 \land n = b \land b \geq 0 \implies P(n, a, b) \]

For \( \mathcal{E}_3 \), we unfold the predicate \( \text{lln} \) pointed by \( \text{root} \) in RHS (recursive case), subtract points-to predicate pointed by \( \text{root} \) in both sides, apply lemma \( \text{jn} \) and generate the following assumption to successfully prove the rest of RHS:

\[ R_3: n = n_1 + 1 \land a = a_1 + 1 \land P(n_1, a_1, b) \land n_1 \geq 0 \land a_1 \geq 0 \land b \geq 0 \implies P(n, a, b) \]

Using a fixed point computation (i.e. FixCalc [33]) to solve \( R_2 \land R_3 \), a definition of \( P \) can be derived as \( P(n, a, b) \equiv n = a + b \land n \geq b \land b \geq 0 \). The lemma \( \text{jn} \) is synthesized as:

\[
\begin{align*}
\text{lemma} \ \text{jn} & \forall n, a, b. \exists p. \text{lseg}n(\text{root}, p, a) \land \text{lln}(p, b) \implies \text{lln}(\text{root}, n) \land n = a + b \land n \geq b \land b \geq 0 \\
\text{lemma} \ \text{sp} & \forall n, a, b. \exists p. \text{lseg}n(\text{root}, p, a) \land \text{lln}(p, b) \\
\end{align*}
\]

Moreover, our system also successfully verifies the lemma \( \text{sp} \) (substituted with \( P \)) as:

\[
\begin{align*}
\text{lemma} \ \text{sp} & \forall n, a, b. \exists p. \text{lseg}n(\text{root}, p, a) \land \text{lln}(p, b) \\
\text{lemma} \ \text{sp} & \forall n, a, b. \text{lln}(\text{root}, n) \land n = a + b \land n \geq b \land b \geq 0 \implies \exists p. \text{lseg}n(\text{root}, p, a) \land \text{lln}(p, b) \\
\end{align*}
\]

Now, \( \text{jn} \) and \( \text{sp} \) can be soundly applied for upcoming proof search. By applying the lemma \( \text{sp} \), our entailment procedure can prove the validity of \( \mathcal{E}_4 \) and inferring the residue as: \( \Delta_{\text{frame}} \equiv \text{emp} \land n \geq k \land k \geq 0 \land i = k \land j = n - k \land i = a' \land j = b' \).

### C.2 Modular Verification with Last Element
To illustrate how our proof system can support induction reasoning together with complex frame inference, consider the modular verification of the method `append_shape` in Fig. 6. This method appends a list pointed by `y` to the end of a list pointed by `x`. The user provides its pre-post specification (lines 2-3) and the predicates `ll`:

\[
\text{pred } ll(\text{root}) \equiv \text{emp} \land \text{root} = \text{null} \\
\lor \exists q \cdot \text{root} \rightarrow c_1(q) \ast ll(q);
\]

As a (bottom-up) modular verification, the loops are verified prior to the verification of the method `check;` and the correctness of a method is reduced to the validity of appropriate verification conditions generated. Our system generates verification conditions to ensure absence of memory errors (no null dereference, no double free and no memory leak), validity of functional calls/loops via compositional pre-/post-conditions and post-conditions holding.

The most challenging step to verify this example is the proving of absence of null dereference at line 6. The symbolic state is computed before line 6 is

\[
ll\_last(t, t') \ast ll(y) \land t \neq \text{null}
\]

For memory safety at line 6, our system generates the following proof obligation

\[
ll\_last(t, t') \ast ll(y) \land t \neq \text{null} \vdash_{\gamma} t' \rightarrow c_1(q)
\]

Since the information of `t'` is deeply embedded in the base case of predicate `ll\_last`, this entail check challenges existing SL proof systems. Additionally for a proper reasoning, a proof system also needs to infer the frame as the list `ll(y)` and a list segment from `t` to the node before `t'`. Inferring such frame is nontrivial. Our system generates the conjecture:

\[
\text{lemma : } ll\_last(t, t') \ast ll(y) \land t \neq \text{null} \rightarrow t' \rightarrow c_1(q) \ast U_2(t, t', q, y)
\]

Then, proves its validity as follows.

\[
\begin{array}{c}
\text{(Base)} \\
ll\_last(t, t') \ast ll(y) \land t \neq \text{null} \vdash_{\gamma} t' \rightarrow c_1(q) \ast U_2(t, t', q, y) \\
\end{array}
\]

\[
\begin{array}{c}
\text{(Induction)} \\
\text{emp} \vdash_{(\ast)} (\sigma_1, \text{emp}) \\
\end{array}
\]

\[
\begin{array}{c}
\text{AF} \\
ll(y) \land t = t' = q = \text{null} \land t \neq \text{null} \vdash c_1(t, t', q, y) \\
\end{array}
\]

\[
\begin{array}{c}
\text{M-} \\
\text{(Base): } t \rightarrow c_1(\text{null}) \ast ll(y) \land t' = t \land t \neq \text{null} \vdash_{\gamma} t' \rightarrow c_1(q) \ast U_2(t, t', q, y)
\end{array}
\]

\[\text{\footnote{Indeed, this invariant is the outcome of the state-of-the-art shape analysis tools like }\textit{#24} \text{ when they are used for invariant inference.}}\]

---

1  `c1* append_shape(c1; x; c1; y)`
2  `//requires ll(x) \ast ll(y) \land x \neq \text{null}`
3  `\text{ensures } ll(\text{res}) + /`
4  `c1; t=x;`
5  `\text{while}(t->next) t=t->next;`
6  `t->next=y;`
7  `\text{return } x;`}

Fig. 6. Code of method `append_shape`.
Relational assumptions are inferred as:

\[ \sigma_1: \ll(y) \land t' = t = q = \text{null} \land t \neq \text{null} \Rightarrow U_2(t, t', q, y) \]

\[ \sigma_2: t \rightarrow c_1(q_1) \rightarrow U_2(q_1, t', q_2, y) \Rightarrow U_2(t, t', q_2, y) \]

From \( \sigma_1 \) and \( \sigma_2 \), our system synthesizes the following definition for \( U_2 \) as

\[ U_2(\text{root}, t', q, y) \equiv \ll(y) \land \text{root} = t' \land q = \text{null} \land \text{root} \neq \text{null} \lor \exists q_1. \text{root} \rightarrow c_1(q_1) \rightarrow U_2(q_1, t', q, y) \]

Using theorem exploration presented in Sec [6] and App. [B], our system generates the following two-way lemma to normalize the predicate \( U_3 \):

\[ \text{lemma consequat} : U_2(\text{root}, t', q, y) \leftrightarrow U_3(\text{root}, t') \land \ll(y) \land q = \text{null} \land \text{root} \neq \text{null} \]

with \( U_3 \) is a newly-inferred predicate as follows.

\[ U_3(\text{root}, t') \equiv \ll(y) \land \text{root} = t' \lor \exists q_1. \text{root} \rightarrow c_1(q_1) \rightarrow U_3(q_1, t') \]

In summary, our system successfully proves validity and infers frame for the entailment.

We present a reasoning on both shape and size properties of a more complicated revision of the append method in App. [C.3]

### C.3 Modular Verification with Incremental Specification Inference

| 1 | \( c_1 \star \text{append(int n, int m)} \) | 7 | \( c_1 \star \text{create_ll(int s)} \) |
|---|---|---|---|
| 2 | \( c_1 \star x = \text{creat_ll}(n) \) | 8 | if (s=0) return null; |
| 3 | \( c_1 \star y = \text{creat_ll}(m) \) | 9 | else { |
| 4 | \( c_1 \star t = x \) | 10 | \( c_1 \star p = \text{malloc}(c_1) \); |
| 5 | while (t->next) t = t->next; | 11 | p->next = \text{create_ll}(s-1); |
| 6 | t->next = y; return x; | 12 | return p; |

**Fig. 7.** Code of method append.

In order to minimize the burden of program verification, we pursue a compositional verification of programs whose specifications are partially supplied. Concretely, the user is only required to provide specifications of pre/post conditions for critical methods and loop invariants whereas specifications of the rest are inferred automatically. This specification inference has been implemented incrementally for the combined domains, such that shape-only specifications are inferred first and then constraints over pure domains are additionally synthesized. In the context of heap-manipulating programs, recursive methods and loop invariants normally relate to recursive predicates; consequently, compositionally verifying these specifications requires both inductive reasoning and frame inference. Our proposed approach brings the best support for such verification.

To illustrate how our proof system is used to compositionally and incrementally verify heap-manipulating programs, consider the method append in Fig. 7 which appends a list pointed by \( y \) to the end of a list pointed by \( x \). The user provides the predicates \( \lln \), \( \lsegm \) (Sec [2]) and append’s specification as: \text{requires } n>0 \land m \geq 0 \text{ ensures } \lln(res, m+n).
Verifying safety properties that require both heap and data reasoning has been studied in the literature, e.g. abstract interpretation \[40\], TVLA \[21\] and interpolation \[2\]. Different to these proposals, ours is compositional and based on the proposed inductive proof system. We enhance inference techniques \[7,24,41\] to generate specification and invariant for the combined domains of the method create, ll and while loop (of pre/post condition). Our verification is bottom-up (i.e. verifying the method create, ll and while loop before the method append) and incremental (i.e. analyzing shape and then size property for the loop invariant inference). Concretely, our system inferred the specification for create, ll and while loop as: requires \(s \geq 0\) ensures lln(res, s) and \((m)\) requires \(\exists q: t \rightarrow c_1(q) \land lln(q, i) \land i \geq 0\) ensures lsegn(t, t', j) \(\rightarrow c_1(\text{null}) \land j = i\), resp.

In the following, we present the specification inference for the loop. To infer shape specification of the loop invariant the system initially generates specification with two unknown shape predicates as follows: requires \(U_{\text{pre}}(t)\) ensures \(U_{\text{post}}(t, t')\), whereas \(t'\) is the value of \(t\) after the loop. Using the modular shape analysis in \[24\], the predicate \(U_{\text{pre}}\) and \(U_{\text{post}}\) are synthesized as

\[
U_{\text{pre}}(\text{root}) \equiv \exists q : \text{root} \rightarrow c_1(q) \land U_1(q)
\]

\[
U_1(\text{root}) \equiv \text{emp} \land \text{root} = \text{null} \lor \exists q : \text{root} \rightarrow c_1(q) \land U_1(q)
\]

\[
U_{\text{post}}(\text{root}, l) \equiv \text{root} \rightarrow c_1(\text{null}) \land \text{root} = l \lor \exists q : \text{root} \rightarrow c_1(q) \land U_{\text{post}}(q, l) \land \text{root} \neq l
\]

whereas \(U_1\) is an auxiliary predicate. Whenever receiving these predicate definitions, our theorem exploration component will generate new lemmas to study interesting properties of these predicates. In this example, our system generates the following lemmas to explicate the separation between two parameters of the predicate \(U_{\text{post}}\):

\[
\text{lemma consep } U_{\text{post}}(\text{root}, l) \leftrightarrow U_2(\text{root}, l) \land l \rightarrow c_1(\text{null})
\]

\[
U_2(\text{root}, s) \equiv \text{emp} \land \text{root} = s \lor \exists q : \text{root} \rightarrow c_1(q) \land U_2(q, s) \land \text{root} \neq s
\]

Then, the shape invariant of the loop is constructed as follows:

\[
\text{requires } \exists q : x \rightarrow c_1(q) \land U_1(q) \quad \text{ensures } U_2(x, x') \land x' \rightarrow c_1(\text{null})
\]

To extend the above shape invariant with the size property, we first use predicate extension \[41\] to automatically append the size property into \(U_1\) (\(U_{\text{nn}}\)) and \(U_2\) (\(U_{\text{nn}}\))

\[
U_{\text{nn}}(\text{root}, n) \equiv \text{emp} \land \text{root} = \text{null} \land n = 0 \lor \exists q : \text{root} \rightarrow c_1(q) \land U_{\text{nn}}(q, n-1)
\]

\[
U_{\text{nn}}(\text{root}, s, n) \equiv \text{emp} \land \text{root} = s \land n = 0 \lor \exists q : \text{root} \rightarrow c_1(q) \land U_{\text{nn}}(q, s, n-1) \land \text{root} \neq s
\]

Our theorem exploration, again, generates two lemmas to match \(U_{\text{nn}}\) with \(\text{lnn}\) and \(U_{\text{nn}}\) with \(\text{lsegn}\) as: \(U_{\text{nn}}(\text{root}, n) \leftrightarrow \text{lln}(\text{root}, n)\) and \(U_{\text{nn}}(\text{root}, s, n) \leftrightarrow \text{lsegn}(\text{root}, s, n)\). After that, we generate specification with unknown pure predicates \(P_2, P_3\):

\[
\text{requires } \exists q : t \rightarrow c_1(q) \land \text{lln}(q, i) \land P_2(i) \quad \text{ensures } \text{lsegn}(t, t', j) \land t' \rightarrow c_1(\text{null}) \land P_2(i, j)
\]

whereas \(P_2\) and \(P_3\) are placeholders to capture constraints over the size variables. Using \(\text{SODB}\) for pure properties inference \[41\], the following definitions are synthesized:

\[
P_2(i) \equiv i \geq 0 \land P_3(i, j) \equiv j = i.
\]

Finally, loop invariant is inferred as the specification \(s_3\). This loop invariant is now used in the verification of the main method append. To verify the correctness and memory safety (no null-dereference and no leakage) of append, our system generates and successfully proves the following three verification conditions:

\[
\text{For simplicity, we discard the verification conditions at line 2 and line 3.}
\]
whereas $L$ is the set of lemmas either supplied by the user or generated by our system (i.e. to explore predicate relations like the `conseq` lemmas above).

We highlight two advantages achieved from our proposed approach. First, if the two-way lemma `conseq` was not synthesized, the condition VC2 would be generated as:

\[
\forall t, t' \vdash \text{lln}(y, m) \land x = t \land t' \neq \text{null} \land n = i - 1 \land n > 0 \land i \geq 0 \]

// before line 6, no null-dereference

VC3. `lseg`n(res, t', j) + t' \rightarrow c_1(y) + \text{lln}(y, m) \land i = j \land x = t \land t' \neq \text{null} \land n = i - 1 \land n > 0 \land m \geq 0

\[
\vdash_L \text{lln}(\text{res}, m + n) \rightarrow (\text{true} \land \text{emp} \land x = t \land t' \neq \text{null} \land n = i - 1 \land n > 0 \land i \geq 0)
\]

// after line 6, post-condition, no leakage

where $L$ is the set of lemmas either supplied by the user or generated by our system

(i.e. to explore predicate relations like the `conseq` lemmas above).

We highlight two advantages achieved from our proposed approach. First, if the two-way lemma `conseq` was not synthesized, the condition VC2 would be generated as:

\[
\forall t, t' \vdash \text{lln}(y, m) \land x = t \land t' \neq \text{null} \land n = i - 1 \land n > 0 \land i \geq 0
\]

// line 5, pre-proving (of loop)