Automated Lemma Synthesis in Symbolic-Heap Separation Logic

QUANG-TRUNG TA*, National University of Singapore, Singapore

TON CHANH LE, National University of Singapore, Singapore

SIAU-CHENG KHOO, National University of Singapore, Singapore

WEI-NGAN CHIN, National University of Singapore, Singapore

The symbolic-heap fragment of separation logic has been actively developed and advocated for verifying the memory-safety property of computer programs. At present, one of its biggest challenges is to effectively prove entailments containing inductive heap predicates. These entailments are usually proof obligations generated when verifying programs that manipulate complex data structures like linked lists, trees, or graphs.

To assist in proving such entailments, this paper introduces a *lemma synthesis framework*, which automatically discovers lemmas to serve as eureka steps in the proofs. Mathematical induction and template-based constraint solving are two pillars of our framework. To derive the supporting lemmas for a given entailment, the framework firstly identifies possible lemma templates from the entailment's heap structure. It then sets up unknown relations among each template's variables and *conducts structural induction proof* to generate constraints about these relations. Finally, it *solves the constraints* to find out actual definitions of the unknown relations, thus discovers the lemmas. We have integrated this framework into a prototype prover and have experimented it on various entailment benchmarks. The experimental results show that our lemma-synthesis-assisted prover can prove many entailments that could not be handled by existing techniques. This new proposal opens up more opportunities to automatically reason with complex inductive heap predicates.

CCS Concepts: • **Theory of computation → Logic and verification**; **Proof theory**; **Automated reasoning**; **Separation logic**; • **Software and its engineering → Software verification**;

Additional Key Words and Phrases: Separation logic, entailment proving, mathematical induction, structural induction, lemma synthesis, proof theory, automated reasoning

ACM Reference Format:

Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2018. Automated Lemma Synthesis in Symbolic-Heap Separation Logic. *Proc. ACM Program. Lang.* 2, POPL, Article 9 (January 2018), 29 pages. https://doi.org/10.1145/3158097

* The first and the second author contributed equally.

Authors' addresses: Quang-Trung Ta, School of Computing, National University of Singapore, Singapore, taqt@comp.nus.edu.sg; Ton Chanh Le, School of Computing, National University of Singapore, Singapore, chanhle@comp.nus.edu.sg; Siau-Cheng Khoo, School of Computing, National University of Singapore, Singapore, khoosc@comp.nus.edu.sg; Wei-Ngan Chin, School of Computing, National University of Singapore, Singapore, chinwn@comp.nus.edu.sg.

2475-1421/2018/1-ART9
https://doi.org/10.1145/3158097

1 INTRODUCTION

Having been actively developed in the recent two decades, separation logic appears as one of the most popular formalisms in verifying the memory-safety property of computer programs [O'Hearn et al. 2001; Reynolds 2002]. It combines *spatial operations*, which describe the separation of the memory, and *inductive heap predicates* to expressively model the shape of complex data structures, such as variants of linked lists, trees, or graphs. This modeling has been successfully realized in both academic research and industrial use. For example, it is implemented by the static analysis tools SLAYer [Berdine et al. 2011] and Infer [Calcagno et al. 2015] to find memory bugs in system code or mobile applications in large scale.

Both SLAYer and Infer employ a logic fragment called *symbolic-heap separation logic*, which differentiates *spatial formulas* describing the memory shape of a program's states from *pure formulas* representing Boolean constraints of the program's variables. This fragment is also utilized by other academic verification systems such as HIP [Chin et al. 2012], jStar [Distefano and Parkinson 2008], and Smallfoot [Berdine et al. 2005a]. Primarily, the verification of a program in such systems involves three phases: (i) specifying desired properties of the program using separation logic, (ii) analyzing the program's behavior against its specification to obtain a set of verification conditions, mostly in the form of *separation logic entailments*, and (iii) proving the collected entailments to determine whether the program behaves accordingly to its specification.

There have been several efforts to prove separation logic entailments, and one of the biggest challenges is to *effectively* handle their inductive heap predicates. A commonly practised approach is to restrict the predicates to only certain classes, such as: predicates whose syntax and semantics are defined beforehand [Berdine et al. 2004, 2005b; Bozga et al. 2010; Pérez and Rybalchenko 2011, 2013; Piskac et al. 2013, 2014], predicates describing only variants of linked lists [Enea et al. 2014], or predicates satisfying the particular *bounded tree width property* [Iosif et al. 2013, 2014]. On the one hand, such predicate restrictions lead to the invention of effective entailment proving techniques; on the other hand, they *prevent* the predicates from modeling complex constraints, which involve not only the shape, but also the size or the content of data structures.

A more general approach to handle inductive heap predicates is to use proof techniques which are *similar to or based on mathematical induction*. These techniques include cyclic proof [Brotherston et al. 2011], structural induction proof [Chu et al. 2015], and mutual induction proof [Ta et al. 2016]. In general, induction-based techniques are capable of reasoning about structures which are recursively defined [Bundy 2001]. Therefore, they can be used to deal with inductive heap predicates in separation logic entailments. However, it is well-known that the induction-based proof techniques are *incomplete*, due to the *failure of cut-elimination* in inductive theories. Consequently, their successes often depend on an eureka step: *discovering supporting lemmas* [Bundy 2001].

Not only are the *supporting lemmas* directly important to *induction-based* proof systems, but they are also highly needed by verification systems that use *non-induction-based* back-end provers, as in the works of Nguyen and Chin [2008] and Qiu et al. [2013]. These systems cannot automatically discover the desired lemmas but *require the users to manually provide* them. Consequently, the aforementioned verification systems are not fully automated.

Although any *valid entailment* can become a lemma, the lemmas that are really important are the ones that can convert between inductive heap predicates, or can combine many inductive heap predicates into another one, or can split an inductive heap predicate into a combination of many others. These lemmas are helpful for *rearranging inductive heap predicates* in a goal entailment so that the entailment proof can be quickly discovered. They are also called *inductive lemmas*, as

induction proofs are often needed to prove them, i.e., there are derivation steps that record and apply induction hypotheses in the lemmas' proofs.

In this work, we propose a novel framework to *automatically synthesize inductive lemmas* to assist in proving entailments in the fragment of symbolic-heap separation logic. Our framework is developed based on structural induction proof and template-based constraint solving. Given a goal entailment, the framework first analyzes the entailment's heap structure to identify essential lemma templates. It then sets up unknown relations among each template's variables and conducts structural induction proof to generate constraints about the relations. Finally, it solves the constraints to find out actual definitions of the unknown relations, thus discovers the desired inductive lemmas.

In summary, our work makes the following contributions:

- We propose a novel framework to automatically synthesize lemmas to assist in proving entailments in the fragment of symbolic-heap separation logic with inductive heap predicates.
- We integrate the lemma synthesis framework into an entailment proof system, which allows the lemmas to be flexibly discovered *on-the-fly* to support the entailment proofs.
- We implement a prototype prover and experiment it with numerous entailments from various benchmarks. The experimental result is promising since our tool can prove most of the valid entailments in these benchmarks and outperforms all existing separation logic provers.

2 MOTIVATING EXAMPLE

In this section, we present a running example to illustrate our work and result. Here, we consider an inductive heap predicate dll and its variant dllrev, both modeling a doubly linked list data structure with a length property.

$$\begin{array}{rcl} \mathsf{dll}(hd,pr,tl,nt,len) & \stackrel{\text{def}}{=} & hd{\mapsto}pr,nt \wedge hd{=}tl \wedge len{=}1 \\ & & \vee\ \exists u.(hd{\mapsto}pr,u * \mathsf{dll}(u,hd,tl,nt,len{-}1)) \\ \mathsf{dllrev}(hd,pr,tl,nt,len) & \stackrel{\text{def}}{=} & hd{\mapsto}pr,nt \wedge hd{=}tl \wedge len{=}1 \\ & & \vee\ \exists u.(\mathsf{dllrev}(hd,pr,u,tl,len{-}1) * tl{\mapsto}u,nt) \end{array}$$

Each element in the linked list is modeled by a singleton heap predicate $x{\mapsto}pr,nt$, where x is its memory address, pr and nt respectively point to the *previous* and the *next* element in the list. Moreover, both $\mathsf{dll}(hd,pr,tl,nt,len)$ and $\mathsf{dllrev}(hd,pr,tl,nt,len)$ denote a non-empty doubly linked list from the first element pointed-to by hd (head) to the last element pointed-to by tl (tail). The *previous* and the *next* element of the entire list are respectively pointed-to by pr and nt, and len denotes the linked list's length. The only difference of the two predicates is that dll is recursively defined from the linked list's head to the tail, whereas dllrev is defined in the reversed direction.

Suppose that when verifying a program manipulating the doubly linked list data structure, a verifier needs to prove an entailment relating to an extraction of the last element from a concatenation of two linked lists with certain constraints on their lengths, such as the following entailment E_1.

$$E_1 \triangleq \mathsf{dllrev}(x,y,u,v,n) * \mathsf{dll}(v,u,z,t,200) \wedge n{\geq}100 \vdash \exists r.(\mathsf{dll}(x,y,r,z,n{+}199) * z{\mapsto}r,t)$$

Unfortunately, the existing entailment proving techniques could not prove E_1. Specifically, the predicate-restriction approaches either consider only singly linked list [Berdine et al. 2005b; Bozga et al. 2010; Cook et al. 2011; Pérez and Rybalchenko 2011, 2013], or do not handle linear arithmetic constraints [Enea et al. 2014; Iosif et al. 2013, 2014], or require a pre-defined translation of heap predicates into other theories [Piskac et al. 2013, 2014]. Moreover, the non-induction-based approaches require users to provide supporting lemmas [Chin et al. 2012]. Finally, the induction-based techniques [Brotherston et al. 2011; Chu et al. 2015] fail to prove E_1 because this entailment is *not general enough* to be an effective induction hypothesis, due to its *specific constant values*.

We briefly explain the *failure of induction proof* on E_1 as follows. Typically, induction is performed on an inductive heap predicate in E_1's antecedent. Suppose the chosen predicate is $\mathsf{dllrev}(x, y, u, v, n)$; E_1 is then recorded as an induction hypothesis H, whose variables are renamed to avoid confusion:

$$H \triangleq \mathsf{dllrev}(a, b, p, q, m) * \mathsf{dll}(q, p, c, d, 200) \wedge m{\geq}100 \vdash \exists k.(\mathsf{dll}(a, b, k, c, m{+}199) * c{\mapsto}k, d)$$

Subsequently, the predicate $\mathsf{dllrev}(x, y, u, v, n)$ of E_1 is unfolded by its recursive definition to derive new sub-goal entailments. We consider an interesting case when the below entailment E_1' is obtained from an inductive case unfolding of $\mathsf{dllrev}(x, y, u, v, n)$:

$$E_1' \triangleq \mathsf{dllrev}(x, y, w, u, n{-}1) * u{\mapsto}w, v * \mathsf{dll}(v, u, z, t, 200) \wedge n{\geq}100 \vdash \exists r.(\mathsf{dll}(x, y, r, z, n{+}199) * z{\mapsto}r, t)$$

When applying the induction hypothesis H to prove E_1', the predicates of the same symbols (dllrev or dll) in the antecedents of H and E_1' need to be unified by a substitution. However, such substitution does not exist since q is mapped to u when unifying $\mathsf{dllrev}(a, b, p, q, m)$ vs. $\mathsf{dllrev}(x, y, w, u, n{-}1)$, whereas q is mapped to the *different* variable v when unifying $\mathsf{dll}(q, p, c, d, 200)$ vs. $\mathsf{dll}(v, u, z, t, 200)$.

Alternatively, we can weaken the spatial formula $u{\mapsto}w, v * \mathsf{dll}(v, u, z, t, 200)$ of E_1''s antecedent into $\mathsf{dll}(u, w, z, t, 201)$, w.r.t. the definition of dll, to derive a new entailment E_1'':

$$E_1'' \triangleq \mathsf{dllrev}(x, y, w, u, n{-}1) * \mathsf{dll}(u, w, z, t, 201) \wedge n{\geq}100 \vdash \exists r.(\mathsf{dll}(x, y, r, z, n{+}199) * z{\mapsto}r, t)$$

Again, no substitution can unify the antecedents of H and E_1''. For example, the substitution $\theta_1 \triangleq [x/a, y/b, w/p, u/q, n{-}1/m]$ might be able to unify $\mathsf{dllrev}(a, b, p, q, m)$ with $\mathsf{dllrev}(x, y, w, u, n{-}1)$, that is, $\mathsf{dllrev}(a, b, p, q, m)\theta_1 \equiv \mathsf{dllrev}(x, y, w, u, n{-}1)$. However, the constraint $n{\geq}100$ in the antecedent of E_1'' *cannot prove* that $n{-}1{\geq}100$, which is obtained from applying θ_1 on the constraint $m{\geq}100$ of H. In addition, the substitution $\theta_2 \triangleq [u/q, w/p, z/c, t/d]$ could not unify $\mathsf{dll}(q, p, c, d, 200)$ with $\mathsf{dll}(u, w, z, t, 201)$ since the two constants 201 and 200 are *non-unifiable*. In short, the above inability in unifying heap predicates makes induction proof fail on E_1.

Nevertheless, the entailment E_1 is provable if necessary lemmas can be discovered. For instance, the following lemmas L_1, L_2, and L_3 can be introduced to assist in proving E_1. More specifically, L_1 *converts* a linked list modeled by the predicate symbol dllrev into a linked list modeled by the variant predicate dll; L_2 *combines* two linked lists modeled by dll into a new one (similar in spirit to the "composition lemma" introduced by Enea et al. [2015]); lastly, L_3 *splits* a linked list modeled by dll into two parts including a new linked list and a singleton heap:

$$\begin{array}{lcl} L_1 & \triangleq & \mathsf{dllrev}(a, b, c, d, m) \vdash \mathsf{dll}(a, b, c, d, m) \\ L_2 & \triangleq & \mathsf{dll}(a, b, p, q, m) * \mathsf{dll}(q, p, c, d, l) \vdash \mathsf{dll}(a, b, c, d, m + l) \\ L_3 & \triangleq & \mathsf{dll}(a, b, c, d, m) \wedge m{\geq}2 \vdash \exists w.(\mathsf{dll}(a, b, w, c, m{-}1) * c{\mapsto}w, d) \end{array}$$

The three lemmas L_1, L_2, and L_3 can be used to prove E_1 as shown in Figure 1. They are successively utilized by the lemma application rules $\mathsf{LM_L}$, $\mathsf{LM_R}$ to finally derive a new entailment E_4, which can be easily proved by standard inference rules in separation logic. We will explain the details of these lemma application rules $\mathsf{LM_L}$, $\mathsf{LM_R}$ and other inference rules in Section 4.1.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{n{\geq}100 \vdash (x{=}x \wedge y{=}y \wedge z{=}z \wedge t{=}t \wedge n{+}200{=}n{+}200)}\ \vdash_{\Pi}}{E_4 \triangleq \mathsf{dll}(x, y, z, t, n{+}200) \wedge n{\geq}100 \vdash \mathsf{dll}(x, y, z, t, n{+}200)}\ *\mathsf{P}}{E_3 \triangleq \mathsf{dll}(x, y, z, t, n{+}200) \wedge n{\geq}100 \vdash \exists r.(\mathsf{dll}(x, y, r, z, n{+}199) * z{\mapsto}r, t)}\ \mathsf{LM_R} \text{ with } L_3 \text{ and } \theta_3}{E_2 \triangleq \mathsf{dll}(x, y, u, v, n) * \mathsf{dll}(v, u, z, t, 200) \wedge n{\geq}100 \vdash \exists r.(\mathsf{dll}(x, y, r, z, n{+}199) * z{\mapsto}r, t)}\ \mathsf{LM_L} \text{ with } L_2 \text{ and } \theta_2}{E_1 \triangleq \mathsf{dllrev}(x, y, u, v, n) * \mathsf{dll}(v, u, z, t, 200) \wedge n{\geq}100 \vdash \exists r.(\mathsf{dll}(x, y, r, z, n{+}199) * z{\mapsto}r, t)}\ \mathsf{LM_L} \text{ with } L_1 \text{ and } \theta_1$$

Fig. 1. Proof tree of E_1 using the lemmas L_1, L_2, and L_3, with the substitutions $\theta_1 \equiv [x/a, y/b, u/c, v/d, n/m]$, $\theta_2 \equiv [x/a, y/b, u/p, v/q, n/m, z/c, t/d, 200/l]$, and $\theta_3 \equiv [x/a, y/b, z/c, t/d, r/w, n{+}200/m]$

In the proof tree in Figure 1, the entailment E_1 can only be concluded valid if all the lemmas L_1, L_2, and L_3 are also valid. Furthermore, this proof tree is not constructed by any induction proofs. Instead, induction is performed in the proof of each lemma. We illustrate a partial induction proof tree of L_1 in Figure 2, where an induction hypothesis is recorded by the induction rule ID and is later utilized by the induction hypothesis application rule IH. As discussed earlier in Section 1, these three lemmas are called *inductive lemmas*. In general, the inductive lemmas help to modularize the proof of a complex entailment like E_1: induction is not directly performed to prove E_1; it is alternatively utilized to prove simpler supporting lemmas such as L_1, L_2, and L_3. This modularity, therefore, increases the success chance of proving complex entailments.

$$\dfrac{\dfrac{\dots}{a{\mapsto}b, d \wedge a{=}c \wedge m{=}1 \vdash \mathsf{dll}(a,b,c,d,m)}\,\mathsf{P_R} \quad \dfrac{\dots}{\mathsf{dllrev}(a,b,u,c,m{-}1) * c{\mapsto}u, d \vdash \mathsf{dll}(a,b,c,d,m)}\,\mathsf{ID}(2)}{\mathsf{dllrev}(a,b,u,c,m{-}1) * c{\mapsto}u, d \vdash \mathsf{dll}(a,b,c,d,m)}\,\mathsf{IH}(1)$$
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{true \vdash \exists t.(v{=}t \wedge a{=}a \wedge c{=}c \wedge d{=}d \wedge m{-}1{=}m{-}1)}\vdash_{\Pi}}{\mathsf{dll}(v,a,c,d,m{-}1) \vdash \exists t.(\mathsf{dll}(t,a,c,d,m{-}1) \wedge v{=}t)}*\mathsf{P}}{\mathsf{dll}(v,a,u,c,m{-}2) * c{\mapsto}u, d \vdash \exists t.(\mathsf{dll}(t,a,c,d,m{-}1) \wedge v{=}t)}\,\mathsf{IH}(2)}{a{\mapsto}b, v * \mathsf{dll}(v,a,u,c,m{-}2) * c{\mapsto}u, d \vdash \exists t.(a{\mapsto}b, t * \mathsf{dll}(t,a,c,d,m{-}1))}*{\mapsto}}{a{\mapsto}b, v * \mathsf{dll}(v,a,u,c,m{-}2) * c{\mapsto}u, d \vdash \mathsf{dll}(a,b,c,d,m)}\,\mathsf{P_R}$$
$$L_1 \triangleq \mathsf{dllrev}(a,b,c,d,m) \vdash \mathsf{dll}(a,b,c,d,m)$$

Fig. 2. Partial induction proof tree of the inductive lemma L_1

where ID(1), ID(2) are performed on $\mathsf{dllrev}(a,b,c,d,m)$, $\mathsf{dll}(a,b,u,c,m{-}1)$ to obtain IHs H_1, H_2, IH(1) with $H_1 \triangleq \mathsf{dllrev}(a',b',c',d',m') \vdash \mathsf{dll}(a',b',c',d',m')$ and $\theta_1 \equiv [a/a', b/b', u/c', c/d', m{-}1/m']$, IH(2) with $H_2 \triangleq \mathsf{dll}(a',b',u',c',m'{-}1){*}c'{\mapsto}u', d' \vdash \mathsf{dll}(a',b',c',d',m')$, $\theta_2 \equiv [v/a', a/b', u/u', c/c', d/d', m{-}1/m']$

On the other hand, we are *not* interested in *trivial lemmas*, which can be proved *directly without using* any induction hypotheses: the rule IH is not used in their proofs. For example, $L_3' \triangleq \mathsf{dll}(a,b,c,d,m) \wedge m{\geq}2 \vdash \exists w.(a{\mapsto}b, w * \mathsf{dll}(w,a,c,d,m{-}1))$ is a trivial lemma, whose direct proof tree is presented in Figure 3. Obviously, if a trivial lemma can be applied to prove an entailment, then the same sequence of inference rules utilized in the lemma's proof can also be directly applied to the goal entailment. For this reason, the *trivial lemmas* do not help to modularize the induction proof.

$$\dfrac{\dfrac{}{a{\mapsto}b, d \wedge a{=}c \wedge m{=}1 \wedge m{\geq}2 \vdash \exists w.(a{\mapsto}b, w{*}\mathsf{dll}(w,a,c,d,m{-}1))}\,\bot^2_{\mathrm{L}} \quad \dfrac{\dfrac{\dfrac{}{m{\geq}2 \vdash \exists w.(u{=}w \wedge a{=}a \wedge c{=}c \wedge d{=}d \wedge m{-}1{=}m{-}1)}\vdash_{\Pi}}{\mathsf{dll}(u,a,c,d,m{-}1) \wedge m{\geq}2 \vdash \exists w.(\mathsf{dll}(w,a,c,d,m{-}1) \wedge u{=}w)}*\mathsf{P}}{a{\mapsto}b, u * \mathsf{dll}(u,a,c,d,m{-}1) \wedge m{\geq}2 \vdash \exists w.(a{\mapsto}b, w * \mathsf{dll}(w,a,c,d,m{-}1))}*{\mapsto}}{L_3' \triangleq \mathsf{dll}(a,b,c,d,m) \wedge m{\geq}2 \vdash \exists w.(a{\mapsto}b, w * \mathsf{dll}(w,a,c,d,m{-}1))}\,\mathsf{ID}$$

Fig. 3. Direct proof tree of the trivial lemma L_3'

In this work, we propose a novel framework to synthesize *inductive lemmas*, such as L_1, L_2, and L_3, to assist in proving separation logic entailments. For a given goal entailment E, we first identify all possible lemma templates, which essentially are entailments constructed from heap predicate symbols appearing in E. The templates will be refined with more Boolean constraints on their variables until *valid* inductive lemmas are discovered. We will explain the details in Section 5.

3 THEORETICAL BACKGROUND

Our lemma synthesis framework is developed to assist in proving entailments in the fragment *of symbolic-heap separation logic* with inductive heap predicates and linear arithmetic. This fragment is similar to those introduced in [Albarghouthi et al. 2015; Brotherston et al. 2011, 2016; Ta et al. 2016]. It is also extended with *unknown relations* to represent Boolean constraints of the desired lemmas. We will present related background of the entailment proof in the following subsections.

3.1 Symbolic-Heap Separation Logic with Unknown Relations

Syntax. We denote our *symbolic-heap separation logic* fragment with *inductive heap predicates* and *unknown relations* as $\text{SL}^{\text{U}}_{\text{ID}}$ and present its syntax in Figure 4. In particular, x is a variable; c and e are respectively an integer constant and an integer expression[1]; nil is a constant denoting a *dangling memory address (null)* and a is a spatial expression modeling a memory address. Moreover, σ denotes a *spatial atom*, which is either (i) a predicate emp modeling an *empty* memory, (ii) a *singleton* heap predicate $x \overset{\iota}{\mapsto} x_1, \ldots, x_n$ describing an n-field data structure of sort ι in the memory, pointed-to by x, and having $x_1, \ldots, x_n$ as values of its fields[2], or (iii) an *inductive* heap predicate $\mathsf{P}(x_1, \ldots, x_n)$ modeling a recursively defined data structure (Definition 3.1). These spatial atoms constitute a *spatial formula* Σ via the separating conjunction operator $*$. On the other hand, π denotes a *pure atom* comprising equality constraints among spatial expressions and linear arithmetic constraints among integer expressions. These pure atoms compose a *pure formula* Π using standard first-order logic connectives and quantifiers. Moreover, Π may contain an *unknown relation* U (Definition 3.4). We will utilize the unknown relation in various phases of the lemma synthesis.

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid -e \mid e_1 + e_2 \mid e_1 - e_2 \mid c\,e \\
a & ::= & \mathsf{nil} \mid x \\
\pi & ::= & \mathit{true} \mid \mathit{false} \mid a_1 = a_2 \mid a_1 \neq a_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 > e_2 \mid e_1 \geq e_2 \mid e_1 < e_2 \mid e_1 \leq e_2 \\
\sigma & ::= & \mathsf{emp} \mid x \overset{\iota}{\mapsto} x_1, \ldots, x_n \mid \mathsf{P}(x_1, \ldots, x_n) \\
\Pi & ::= & \pi \mid U(x_1, \ldots, x_n) \mid \neg\Pi \mid \Pi_1 \wedge \Pi_2 \mid \Pi_1 \vee \Pi_2 \mid \Pi_1 \rightarrow \Pi_2 \mid \forall x.\Pi \mid \exists x.\Pi \\
\Sigma & ::= & \sigma \mid \Sigma_1 * \Sigma_2 \\
F & ::= & \Sigma \mid \Pi \mid \Sigma \wedge \Pi \mid \exists x.\, F
\end{array}
$$

Fig. 4. Syntax of formulas in $\text{SL}^{\text{U}}_{\text{ID}}$

Definition 3.1 (Inductive heap predicate). [Ta et al. 2016] A system of k inductive heap predicates P_i of arity n_i, with $i = 1, \ldots, k$, is defined as follows:

$$
\left\{ \mathsf{P}_i(x^i_1, \ldots, x^i_{n_i}) \quad \overset{\text{def}}{=} \quad F^i_1(x^i_1, \ldots, x^i_{n_i}) \ \vee \ \ldots \ \vee \ F^i_{m_i}(x^i_1, \ldots, x^i_{n_i}) \right\}^k_{i=1}
$$

where F^i_j is a *definition case* of P_i. This fact is also denoted by $F^i_j(x^i_1, \ldots, x^i_{n_i}) \overset{\text{def}}{\Rightarrow} \mathsf{P}_i(x^i_1, \ldots, x^i_{n_i})$, with $1 \leq j \leq m_i$. Moreover, F^i_j is a *base case* if it does not contain any predicates recursively defined with P_i; otherwise, it is an *inductive case*.

Example 3.2. The two predicates $\mathsf{dll}(hd, pr, tl, nt, len)$ and $\mathsf{dllrev}(hd, pr, tl, nt, len)$ in Section 2 are two examples of inductive heap predicates. Moreover, their definitions are *self-recursively* defined.

Example 3.3. The two predicates ListE and ListO [Brotherston et al. 2011] are *mutually recursively defined* to model linked list segments which respectively contain *even* and *odd* number of elements.

$$
\text{(1) } \mathsf{ListO}(x, y) \overset{\text{def}}{=} x \mapsto y \ \vee \ \exists u.(x \mapsto u * \mathsf{ListE}(u, y)) \qquad \text{(2) } \mathsf{ListE}(x, y) \overset{\text{def}}{=} \exists u.(x \mapsto u * \mathsf{ListO}(u, y))
$$

Definition 3.4 (Unknown relation). An unknown relation $U(u_1, \ldots, u_n)$ is an n-ary pure predicate in first-order logic, whose definition is undefined.

Semantics. Figure 5 exhibits the semantics of formulas in $\text{SL}^{\text{U}}_{\text{ID}}$. Given a set Var of variables, Sort of sorts, Val of values, Loc of memory addresses (Loc $\subset$ Val), a model of a formula consists of: a *stack* model s, which is a function s: Var $\rightarrow$ Val, and a *heap* model h, which is a partial function h: (Loc $\times$ Sort) $\rightharpoonup$ Val$^+$. We write $[\![\Pi]\!]_s$ to denote the valuation of a pure formula Π under the stack model s. Moreover, dom(h) denotes the domain of h; $h \ \# \ h'$ indicates that h and h' have

[1] We write $c\,e$ to denote the multiplication by constant in linear arithmetic.

[2] Each *sort* ι represents a unique data type. For brevity, we omit using it when presenting the motivating example.

disjoint domains, i.e., $\mathrm{dom}(h) \cap \mathrm{dom}(h') = \varnothing$; and $h \circ h'$ is the union of two disjoint heap models h and h'. Besides, $[f \mid x{:}y]$ is a function like f except that it returns y for the input x. To define the semantics of the inductive heap predicates, we follow the standard *least fixed point semantics* by interpreting an inductive predicate symbol P as the least prefixed point $[\![\mathsf{P}]\!]$ of a monotone operator constructed from its inductive definition. The construction is standard and can be found in many places, such as [Brotherston and Simpson 2011].

$$
\begin{array}{lll}
s, h \vDash \Pi & \text{iff} & [\![\Pi]\!]_s = \mathit{true} \text{ and } \mathrm{dom}(h) = \varnothing \\
s, h \vDash \mathrm{emp} & \text{iff} & \mathrm{dom}(h) = \varnothing \\
s, h \vDash x \overset{\iota}{\mapsto} x_1, ..., x_n & \text{iff} & \mathrm{dom}(h) = \{s(x)\} \text{ and } h(s(x), \iota) = (s(x_1), ..., s(x_n)) \\
s, h \vDash \mathsf{P}(x_1, ..., x_n) & \text{iff} & (h, [\![x_1]\!]_s, ..., [\![x_n]\!]_s) \in [\![\mathsf{P}]\!] \\
s, h \vDash \Sigma_1 * \Sigma_2 & \text{iff} & \exists h_1, h_2 : h_1 \# h_2 \text{ and } h_1 \circ h_2 = h \text{ and } s, h_1 \vDash \Sigma_1 \text{ and } s, h_2 \vDash \Sigma_2 \\
s, h \vDash \Sigma \wedge \Pi & \text{iff} & [\![\Pi]\!]_s = \mathit{true} \text{ and } s, h \vDash \Sigma \\
s, h \vDash \exists x.\, F & \text{iff} & \exists v \in \mathsf{Val} : [s \mid x{:}v], h \vDash F
\end{array}
$$

Fig. 5. Semantics of formulas in $\mathrm{SL}^{\cup}_{\mathrm{ID}}$

Entailments. Given the syntax and semantics of formulas, we can define entailments as follows. This definition is similar to those in separation logic's literature, such as [Berdine et al. 2004].

Definition 3.5 (Entailment). An entailment between two formulas F_1 and F_2, denoted as $F_1 \vdash F_2$, is said to be *valid*, iff $s, h \vDash F_1$ implies that $s, h \vDash F_2$, for all models s, h. Formally,

$$F_1 \vdash F_2 \text{ is valid,} \quad \text{iff} \quad \forall s, h.\, (s, h \vDash F_1 \rightarrow s, h \vDash F_2)$$

Here, F_1 and F_2 are respectively called the antecedent and the consequent of the entailment. In general, separation logic entailments satisfy a *transitivity property* as stated in Theorem 3.6.

Theorem 3.6 (Entailment transitivity). *Given two entailments $F_1 \vdash \exists \vec{x}.F_2$ and $F_2 \vdash F_3$, where $\vec{x}$ is a list of variables. If both of them are valid, then the entailment $F_1 \vdash \exists \vec{x}.F_3$ is also valid.*

Proof. Consider an arbitrary model s, h such that $s, h \vDash F_1$. Since $F_1 \vdash \exists \vec{x}.F_2$ is valid, it follows that $s, h \vDash \exists \vec{x}.F_2$. By the semantics of $\mathrm{SL}^{\cup}_{\mathrm{ID}}$'s formulas, s can be extended with values $\vec{v}$ of $\vec{x}$ to obtain $s' = [s \mid \vec{x}{:}\vec{v}]$ such that $s', h \vDash F_2$. Since $F_2 \vdash F_3$ is valid, it follows that $s', h \vDash F_3$. Since $s' = [s \mid \vec{x}{:}\vec{v}]$, it is implied by the semantics of $\mathrm{SL}^{\cup}_{\mathrm{ID}}$'s formulas again that $s, h \vDash \exists \vec{x}.F_3$. We have shown that for an arbitrary model s, h, if $s, h \vDash F_1$, then $s, h \vDash \exists \vec{x}.F_3$. Therefore, $F_1 \vdash \exists \vec{x}.F_3$ is valid. □

Substitution. We write $[e_1/v_1, \ldots, e_n/v_n]$, or θ for short, to denote a simultaneous substitution; and $F[e_1/v_1, \ldots, e_n/v_n]$ denotes a formula that is obtained from F by *simultaneously replacing* all occurrences of the variables $v_1, \ldots, v_n$ by the expressions $e_1, \ldots, e_n$. The simultaneous substitution, or substitution for short, has the following properties, given that $\mathsf{FV}(F)$ and $\mathsf{FV}(e)$ are lists of all free variables respectively occurring in the formula F and the expression e.

Proposition 3.7 (Substitution law for formulas [Reynolds 2008]). *Given a separation logic formula F and a substitution $\theta \equiv [e_1/v_1, \ldots, e_n/v_n]$. Let s, h be a separation logic model, where* $\mathrm{dom}(s)$ *contains* $(\mathsf{FV}(F) \setminus \{v_1, \ldots, v_n\}) \cup \mathsf{FV}(e_1) \cup \ldots \cup \mathsf{FV}(e_n)$*, and let $\hat{s} = [s \mid v_1 : [\![e_1]\!]_s \mid \ldots \mid v_n : [\![e_n]\!]_s]$. Then $s, h \vDash F\theta$ if and only if $\hat{s}, h \vDash F$*

Theorem 3.8 (Substitution law for entailments). *Given a separation logic entailment $F_1 \vdash F_2$ and a substitution θ. If $F_1 \vdash F_2$ is valid, then $F_1\theta \vdash F_2\theta$ is also valid.*

Proof. Suppose that $\theta \equiv [e_1/v_1, \ldots, e_n/v_n]$. Consider an arbitrary model s, h such that $s, h \vDash F_1\theta$. Let $\hat{s} = [s \mid v_1 : [\![e_1]\!]_s \mid \ldots \mid v_n : [\![e_n]\!]_s]$. By Proposition 3.7, $s, h \vDash F_1\theta$ implies that $\hat{s}, h \vDash F_1$. Since $F_1 \vdash F_2$ is valid, it follows that $\hat{s}, h \vDash F_2$. By Proposition 3.7 again, $s, h \vDash F_2\theta$. We have shown that for an arbitrary model s, h, if $s, h \vDash F_1\theta$, then $s, h \vDash F_2\theta$. Therefore, $F_1\theta \vdash F_2\theta$ is valid. □

Unknown Entailments and Unknown Assumptions. Recall that we propose to synthesize lemmas by firstly discovering essential lemma templates and then refining them. The template refinement is conducted in 3 steps: (i) creating *unknown entailments*, which contain *unknown relations* representing the lemmas' desired pure constraints, (ii) proving the entailments by induction and collecting *unknown assumptions* about the relations, and (iii) solving the assumptions to discover the lemmas. We formally define the *unknown entailments* and the *unknown assumptions* as follows.

Definition 3.9 (Unknown entailment). An entailment $F_1 \vdash F_2$ is called an *unknown entailment* if the antecedent F_1 or the consequent F_2 contains an unknown relation $U(\vec{x})$.

Definition 3.10 (Unknown assumption). A pure implication $\Pi_1 \rightarrow \Pi_2$ is called an *unknown assumption* of the unknown relation $U(\vec{x})$ if $U(\vec{x})$ occurs in at least one of the two pure formulas Π_1 and Π_2.

Syntactic Equivalence. An entailment induction proof often contains a step that finds a substitution to unify the antecedents of an induction hypothesis and of the goal entailment. In this work, we syntactically check the unification between two spatial formulas by using a *syntactic equivalence* relation defined earlier in [Ta et al. 2016]. We formally restate this relation in the below.

Definition 3.11 (Syntactic equivalence [Ta et al. 2016]). The syntactical equivalence relation of two spatial formulas Σ_1 and Σ_2, denoted as $\Sigma_1 \cong \Sigma_2$, is inductively defined as follows:

(1) $\mathsf{emp} \cong \mathsf{emp}$ (2) $u \overset{\iota}{\mapsto} v_1, \ldots, v_n \cong u \overset{\iota}{\mapsto} v_1, \ldots, v_n$ (3) $\mathsf{P}(u_1, \ldots, u_n) \cong \mathsf{P}(u_1, \ldots, u_n)$

(4) $(\Sigma_1 \cong \Sigma_1') \wedge (\Sigma_2 \cong \Sigma_2') \rightarrow (\Sigma_1 * \Sigma_2 \cong \Sigma_1' * \Sigma_2') \wedge (\Sigma_1 * \Sigma_2 \cong \Sigma_2' * \Sigma_1')$

3.2 Structural Induction Proof for Separation Logic Entailments

We develop the lemma synthesis framework based on a standard structural induction proof. This proof technique is an instance of *Noetherian induction*, a.k.a. well-founded induction [Bundy 2001]. We will briefly explain both Noetherian induction and structural induction here. Nonetheless, our lemma synthesis idea can also be integrated into other induction-based proof techniques.

Noetherian Induction. Given $\mathcal{P}$ is a conjecture on structures of type τ where $\prec_\tau$ is a *well-founded relation* among these structures, i.e., there is no infinite chain like $\ldots \prec_\tau \alpha_n \prec_\tau \ldots \prec_\tau \alpha_1$. Then the *Noetherian induction principle* states that: $\mathcal{P}$ is said to hold for all these structures, if for any structure α, the fact that $\mathcal{P}(\beta)$ holds for all structures $\beta \prec_\tau \alpha$ implies that $\mathcal{P}(\alpha)$ also holds. Formally:

$$\frac{\forall \alpha : \tau.\ (\forall \beta : \tau \prec_\tau \alpha.\ \mathcal{P}(\beta)) \rightarrow \mathcal{P}(\alpha)}{\forall \alpha : \tau.\ \mathcal{P}(\alpha)}$$

Substructural Relation. We prove entailments by applying Noetherian induction on the structure of inductive heap predicates. The *substructural relation* is formally defined in Definition 3.12.

Definition 3.12 (Substructural relation). A heap predicate $\mathsf{P}_1(\vec{v})$ is said to be a *substructure* of $\mathsf{P}_2(\vec{u})$, denoted by $\mathsf{P}_1(\vec{v}) \prec \mathsf{P}_2(\vec{u})$, if $\mathsf{P}_1(\vec{v})$ occurs in a formula obtained from *directly unfolding* $\mathsf{P}_2(\vec{u})$ or from *unfolding any substructure* of $\mathsf{P}_2(\vec{u})$. These conditions are formally stated as follows:

(1) $\exists \vec{w}, \Sigma, \Pi, F(\vec{u}).\ (F(\vec{u}) \cong \exists \vec{w}.(\mathsf{P}_1(\vec{v}) * \Sigma \wedge \Pi) \ \wedge \ F(\vec{u}) \overset{\text{def}}{\Rightarrow} \mathsf{P}_2(\vec{u}))$

(2) $\exists \vec{w}, \Sigma, \Pi, F(\vec{t}), \mathsf{P}'(\vec{t}).\ (F(\vec{t}) \cong \exists \vec{w}.(\mathsf{P}_1(\vec{v}) * \Sigma \wedge \Pi) \ \wedge \ F(\vec{t}) \overset{\text{def}}{\Rightarrow} \mathsf{P}'(\vec{t}) \ \wedge \ \mathsf{P}'(\vec{t}) \prec \mathsf{P}_2(\vec{u}))$

In the above definition, P_1 and P_2 can be the same or different predicate symbols. The latter happens when they are mutually recursively defined, such as ListE and ListO in Example 3.3.

Theorem 3.13 (Well-foundedness). *Given an inductive heap predicate $\mathsf{P}_1(\vec{u}_1)$. If it is satisfiable, i.e., $\mathsf{P}_1(\vec{u}_1) \not\equiv$ false, then under the least fixed point semantics of inductive heap predicates, there is no infinite chain like $\ldots \prec \mathsf{P}_n(\vec{u}_n) \prec \ldots \prec \mathsf{P}_1(\vec{u}_1)$.*

Proof. Suppose that there exists an infinite chain $\ldots \prec \mathsf{P}_n(\vec{u}_n) \prec \ldots \prec \mathsf{P}_1(\vec{u}_1)$. For all $i \geq 1$, we can always insert all predicates derived when unfolding $\mathsf{P}_i(\vec{u}_i)$ to obtain $\mathsf{P}_{i+1}(\vec{u}_{i+1})$ into the current

chain. Therefore, w.l.o.g., we can assume that $\mathsf{P}_{i+1}(\vec{u}_{i+1})$ is obtained from directly unfolding $\mathsf{P}_i(\vec{u}_i)$, for all $i \geq 1$. Hence, $\ldots \prec \mathsf{P}_n(\vec{u}_n) \prec \ldots \prec \mathsf{P}_1(\vec{u}_1)$ is also the infinite unfolding chain of $\mathsf{P}_1(\vec{u}_1)$. In the *least fixed point semantics*, if a predicate is unfolded infinitely, it can be evaluated to *false*. Therefore, $\mathsf{P}_1(\vec{u}_1) \equiv \mathit{false}$. This contradicts with the theorem's hypothesis that $\mathsf{P}_1(\vec{u}_1)$ is satisfiable. □

Structural Induction. We apply the *substructural relation* $\prec$ to propose a *structural induction principle* for the entailment proof. Consider an entailment E whose antecedent contains an inductive predicate $\mathsf{P}(\vec{u})$, that is, $E \triangleq F_1 * \mathsf{P}(\vec{u}) \vdash F_2$, for some formulas F_1, F_2. We write $E(\mathsf{P}(\vec{u}))$ to parameterize E by $\mathsf{P}(\vec{u})$; and $E(\mathsf{P}(\vec{v}))$ is an entailment obtained from $E(\mathsf{P}(\vec{u}))$ by replacing $\mathsf{P}(\vec{u})$ by $\mathsf{P}(\vec{v})$ and respectively replacing variables in $\vec{u}$ by variables in $\vec{v}$. Moreover, we write $\models E(\mathsf{P}(\vec{u}))$ to denote that E holds for $\mathsf{P}(\vec{u})$. The structural induction principle is formally stated as follows:

Theorem 3.14 (Structural induction). *The entailment $E(\mathsf{P}(\vec{u})) \triangleq F_1 * \mathsf{P}(\vec{u}) \vdash F_2$ is valid, if for all predicate $\mathsf{P}(\vec{u})$, the fact that E holds for all sub-structure predicates $\mathsf{P}(\vec{v})$ of $\mathsf{P}(\vec{u})$ implies that E also holds for $\mathsf{P}(\vec{u})$. Formally:*

$$\frac{\forall \mathsf{P}(\vec{u}).\ (\forall \mathsf{P}(\vec{v}) \prec \mathsf{P}(\vec{u}).\ \models E(\mathsf{P}(\vec{v}))) \rightarrow \models E(\mathsf{P}(\vec{u}))}{\forall \mathsf{P}(\vec{u}).\ \models E(\mathsf{P}(\vec{u}))}$$

Proof. We consider two scenarios w.r.t. $\mathsf{P}(\vec{u})$ in the entailment $E(\mathsf{P}(\vec{u})) \triangleq F_1 * \mathsf{P}(\vec{u}) \vdash F_2$ as follows. (1) If $\mathsf{P}(\vec{u}) \equiv \mathit{false}$. Then $F_1 * \mathsf{P}(\vec{u}) \equiv \mathit{false}$, therefore $E(\mathsf{P}(\vec{u}))$ is valid. (2) If $\mathsf{P}(\vec{u}) \not\equiv \mathit{false}$. Then by Theorem 3.13, the substructural relation $\prec$, which applies to $\mathsf{P}(\vec{u})$, is *well-founded*. In this scenario, the above induction principle is an instance of the Noetherian induction [Bundy 2001] where the *substructural relation* $\prec$ is chosen as the *well-founded relation*. Therefore, the correctness of this theorem is automatically implied by the soundness of the Noetherian induction principle. □

4 THE STRUCTURAL INDUCTION PROOF SYSTEM

In this section, we present a structural induction proof system for separation logic entailments. Initially, this system aims to prove *normal entailments* using a set of inference rules (Section 4.1). These rules include the two lemma application rules which apply synthesized lemmas to assist in proving entailments (Figure 8). We use the same proof system to reason about *unknown entailments*, introduced in the lemma synthesis, using a set of synthesis rules (Section 4.2). The proof system also includes a proof search procedure which selectively applies the aforementioned rules to prove a goal entailment (Section 4.3). We will explain the details in the following sections.

4.1 Inference Rules for Standard Entailments

Each inference rule of our proof system contains zero or more premises, a conclusion, and possibly a side condition. The premises and the conclusion are in the form of $\mathcal{H},\ \mathcal{L},\ F_1 \vdash F_2$, where $\mathcal{H}$ and $\mathcal{L}$ are respectively sets of induction hypotheses and valid lemmas, and $F_1 \vdash F_2$ is the (sub-)goal entailment. An inference rule can be interpreted as follows: *if all entailments in its premises are valid, and its side condition (if present) is satisfied, then its goal entailment is also valid.*

For brevity, we write F to denote a symbolic-heap formula $\exists \vec{x}.(\Sigma \wedge \Pi)$, where $\vec{x}$ is a list of quantified variables (possibly empty). Furthermore, we define $F * \Sigma' \triangleq \exists \vec{x}.(\Sigma * \Sigma' \wedge \Pi)$ and $F \wedge \Pi' \triangleq \exists \vec{x}.(\Sigma \wedge \Pi \wedge \Pi')$, given that $\mathsf{FV}(\Sigma') \cap \vec{x} = \varnothing$ and $\mathsf{FV}(\Pi') \cap \vec{x} = \varnothing$. We also write $\vec{u} = \vec{v}$ to denote $(u_1 = v_1) \wedge \ldots \wedge (u_n = v_n)$, given that $\vec{u} \triangleq u_1, \ldots, u_n$ and $\vec{v} \triangleq v_1, \ldots, v_n$ are two variable lists of the same size. Finally, $\vec{u} \# \vec{v}$ indicates that the two lists $\vec{u}$ and $\vec{v}$ are disjoint, i.e., $\nexists w.(w \in \vec{u} \wedge w \in \vec{v})$.

The set of inference rules comprises logical rules (Figure 6) reasoning about the logical structure of the entailments, induction rules (Figure 7) proving the entailments by structural induction, and lemma application rules (Figure 8) applying supporting lemmas to assist in proving the entailments.

$$\bot^1_{\mathrm{L}}\ \frac{}{\mathcal{H},\ \mathcal{L},\ F_1 * u\overset{\iota_1}{\mapsto}\vec{v} * u\overset{\iota_2}{\mapsto}\vec{t} \vdash F_2} \qquad \bot^2_{\mathrm{L}}\ \frac{}{\mathcal{H},\ \mathcal{L},\ F_1 \wedge \Pi_1 \vdash F_2}\ \Pi_1 \rightarrow \mathit{false} \qquad \vdash_{\Pi}\ \frac{}{\mathcal{H},\ \mathcal{L},\ \Pi_1 \vdash \Pi_2}\ \Pi_1 \rightarrow \Pi_2$$

$$=_{\mathrm{L}}\ \frac{\mathcal{H},\ \mathcal{L},\ F_1[u/v] \vdash F_2[u/v]}{\mathcal{H},\ \mathcal{L},\ F_1 \wedge u{=}v \vdash F_2} \qquad \mathsf{E}_{\mathrm{L}}\ \frac{\mathcal{H},\ \mathcal{L},\ F_1 \vdash F_2}{\mathcal{H},\ \mathcal{L},\ F_1 * \mathrm{emp} \vdash F_2} \qquad \mathsf{E}_{\mathrm{R}}\ \frac{\mathcal{H},\ \mathcal{L},\ F_1 \vdash \exists \vec{x}.F_2}{\mathcal{H},\ \mathcal{L},\ F_1 \vdash \exists \vec{x}.(F_2 * \mathrm{emp})}$$

$$\exists_{\mathrm{L}}\ \frac{\mathcal{H},\ \mathcal{L},\ F_1[u/x] \vdash F_2}{\mathcal{H},\ \mathcal{L},\ \exists x.F_1 \vdash F_2}\ u \notin \mathsf{FV}(F_2) \qquad \mathsf{CA}\ \frac{\mathcal{H},\ \mathcal{L},\ F_1 \wedge \Pi \vdash F_2 \qquad \mathcal{H},\ \mathcal{L},\ F_1 \wedge \neg\Pi \vdash F_2}{\mathcal{H},\ \mathcal{L},\ F_1 \vdash F_2}$$

$$\exists_{\mathrm{R}}\ \frac{\mathcal{H},\ \mathcal{L},\ F_1 \vdash \exists \vec{x}.F_2[u/v]}{\mathcal{H},\ \mathcal{L},\ F_1 \vdash \exists \vec{x}, v.(F_2 \wedge u{=}v)} \qquad *{\mapsto}\ \frac{\mathcal{H},\ \mathcal{L},\ F_1 \vdash \exists \vec{x}.(F_2 \wedge u{=}t \wedge \vec{v}{=}\vec{w})}{\mathcal{H},\ \mathcal{L},\ F_1 * u\overset{\iota}{\mapsto}\vec{v} \vdash \exists \vec{x}.(F_2 * t\overset{\iota}{\mapsto}\vec{w})}\ u \notin \vec{x},\ \vec{v}\ \#\ \vec{x}$$

$$\mathsf{P}_{\mathrm{R}}\ \frac{\mathcal{H},\ \mathcal{L},\ F_1 \vdash \exists \vec{x}.(F_2 * F_i^{\mathsf{P}}(\vec{u}))}{\mathcal{H},\ \mathcal{L},\ F_1 \vdash \exists \vec{x}.(F_2 * \mathsf{P}(\vec{u}))}\ F_i^{\mathsf{P}}(\vec{u}) \overset{\text{def}}{\Rightarrow} \mathsf{P}(\vec{u}) \qquad *\mathsf{P}\ \frac{\mathcal{H},\ \mathcal{L},\ F_1 \vdash \exists \vec{x}.(F_2 \wedge \vec{u}{=}\vec{v})}{\mathcal{H},\ \mathcal{L},\ F_1 * \mathsf{P}(\vec{u}) \vdash \exists \vec{x}.(F_2 * \mathsf{P}(\vec{v}))}\ \vec{u}\ \#\ \vec{x}$$

Fig. 6. Logical rules

Logical Rules. The set of logical rules in Figure 6 consists of:

- *Axiom rules* $\vdash_{\Pi}, \bot^1_{\mathrm{L}}, \bot^2_{\mathrm{L}}$. These rules prove pure entailments by invoking an off-the-shelf prover such as Z3 [Moura and Bjørner 2008] (as in the rule $\vdash_{\Pi}$), or prove entailments whose antecedents are inconsistent, i.e., they contain overlaid singleton heaps ($u\overset{\iota_1}{\mapsto}\vec{v}$ and $u\overset{\iota_2}{\mapsto}\vec{t}$ in the rule $\bot^1_{\mathrm{L}}$) or contradictions ($\Pi_1 \rightarrow \mathit{false}$ in the rule $\bot^2_{\mathrm{L}}$).
- *Normalization rules* $\exists_{\mathrm{L}}, \exists_{\mathrm{R}}, =_{\mathrm{L}}, \mathsf{E}_{\mathrm{L}}, \mathsf{E}_{\mathrm{R}}$. These rules simplify the goal entailment by either eliminating existentially quantified variables ($\exists_{\mathrm{L}}, \exists_{\mathrm{R}}$), or removing equalities ($=_{\mathrm{L}}$) or empty heap predicates ($\mathsf{E}_{\mathrm{L}}, \mathsf{E}_{\mathrm{R}}$) from the entailment.
- *Case analysis rule* CA. This rule performs a case analysis on a pure condition Π by deriving two sub-goal entailments whose antecedents respectively contain Π and $\neg\Pi$. The underlying principle of this rule is known as the *law of excluded middle* [Whitehead and Russell 1912].
- *Unfolding rule* P_{R}. This rule derives a new entailment by unfolding a heap predicate in the goal entailment's consequent by its inductive definition.
- *Matching rules* $*{\mapsto}, *\mathsf{P}$. These rules remove *identical* singleton heap predicates ($*{\mapsto}$) or inductive heap predicates ($*\mathsf{P}$) from two sides of the goal entailments. Here, we ensure that these predicates are identical by adding equality constraints about their parameters into the derived entailments' consequents.

$$\mathsf{ID}\ \frac{\mathcal{H}',\ \mathcal{L},\ \Sigma_1 * F_1^{\mathsf{P}}(\vec{u}) \wedge \Pi_1 \vdash F_2 \qquad \ldots \qquad \mathcal{H}',\ \mathcal{L},\ \Sigma_1 * F_m^{\mathsf{P}}(\vec{u}) \wedge \Pi_1 \vdash F_2}{\mathcal{H},\ \mathcal{L},\ \Sigma_1 * \mathsf{P}(\vec{u}) \wedge \Pi_1 \vdash F_2}\ \mathsf{P}(\vec{u}) \overset{\text{def}}{=} F_1^{\mathsf{P}}(\vec{u}) \vee \ldots \vee F_m^{\mathsf{P}}(\vec{u});\ \dagger$$

$\dagger : \mathcal{H}' \triangleq \mathcal{H} \cup \{H\}$, where H is obtained by freshly renaming all variables of $\Sigma_1 * \mathsf{P}(\vec{u}) \wedge \Pi_1 \vdash F_2$

$$\mathsf{IH}\ \frac{\mathcal{H} \cup \{\Sigma_3 * \mathsf{P}(\vec{v}) \wedge \Pi_3 \vdash F_4\},\ \mathcal{L},\ F_4\theta * \Sigma \wedge \Pi_1 \vdash F_2}{\mathcal{H} \cup \{\Sigma_3 * \mathsf{P}(\vec{v}) \wedge \Pi_3 \vdash F_4\},\ \mathcal{L},\ \Sigma_1 * \mathsf{P}(\vec{u}) \wedge \Pi_1 \vdash F_2}\ \begin{array}{l}\mathsf{P}(\vec{u}) \prec \mathsf{P}(\vec{v});\\ \exists\theta, \Sigma.(\Sigma_1 * \mathsf{P}(\vec{u}) \ \cong\ \Sigma_3\theta * \mathsf{P}(\vec{v})\theta * \Sigma) \wedge (\Pi_1 \rightarrow \Pi_3\theta)\end{array}$$

Fig. 7. Induction rules

Induction Rules. The structural induction principle is integrated into our proof system via the two induction rules ID and IH (Figure 7). These rules are explained in details as follows:

- *Rule* ID. This rule performs the structural induction proof on a chosen heap predicate $\mathsf{P}(\vec{u})$ in the antecedent of the goal entailment $\Sigma_1 * \mathsf{P}(\vec{u}) \wedge \Pi_1 \vdash F_2$. In particular, the predicate $\mathsf{P}(\vec{u})$ is unfolded by its inductive definition $\mathsf{P}(\vec{u}) \overset{\text{def}}{=} F_1^{\mathsf{P}}(\vec{u}) \vee \ldots \vee F_m^{\mathsf{P}}(\vec{u})$ to derive new sub-goal entailments as shown in this rule's premises. Moreover, the goal entailment is also recorded as an induction hypothesis in the set $\mathcal{H}'$ so that it can be utilized later to prove the sub-goal entailments.

- *Rule* IH. This rule applies an appropriate recorded induction hypothesis $H \triangleq \Sigma_3 * \mathsf{P}(\vec{v}) \wedge \Pi_3 \vdash F_4$ to prove the goal entailment $\Sigma_1 * \mathsf{P}(\vec{u}) \wedge \Pi_1 \vdash F_2$. It first checks the well-founded condition $\mathsf{P}(\vec{u}) \prec \mathsf{P}(\vec{v})$, i.e. $\mathsf{P}(\vec{u})$ is a substructure of $\mathsf{P}(\vec{v})$, which is required by the structural induction principle. In practice, this condition can be easily examined by labeling each inductive heap predicate $\mathsf{Q}(\vec{x})$ in a proof tree with a set of its ancestor predicates, which are consecutively unfolded to derive $\mathsf{Q}(\vec{x})$. By that mean, $\mathsf{P}(\vec{u}) \prec \mathsf{P}(\vec{v})$ iff $\mathsf{P}(\vec{v})$ appears in the label of $\mathsf{P}(\vec{u})$. Afterwards, the induction hypothesis application is performed in two steps:

  – *Unification step:* unify the antecedents of both the goal entailment and the induction hypothesis by syntactically finding a substitution θ and a spatial formula Σ such that $\Sigma_1 * \mathsf{P}(\vec{u}) \cong \Sigma_3\theta * \mathsf{P}(\vec{v})\theta * \Sigma$ and $\Pi_1 \rightarrow \Pi_3\theta$. If these conditions hold, then it is certain that the entailment $E \triangleq \Sigma_1 * \mathsf{P}(\vec{u}) \wedge \Pi_1 \vdash \Sigma_3\theta * \mathsf{P}(\vec{v})\theta * \Sigma \wedge \Pi_3\theta \wedge \Pi_1$ is valid.

  – *Proving step:* if such θ and Σ exist, then derive a new sub-goal entailment $F_4\theta * \Sigma \wedge \Pi_1 \vdash F_2$. We will explain why this sub-goal entailment is derived. The induction hypothesis $\Sigma_3 * \mathsf{P}(\vec{v}) \wedge \Pi_3 \vdash F_4$ implies that $H\theta \triangleq \Sigma_3\theta * \mathsf{P}(\vec{v})\theta \wedge \Pi_3\theta \vdash F_4\theta$ is also valid, by Theorem 3.8. From E and $H\theta$, we then have a derivation chain: $\Sigma_1 * \mathsf{P}(\vec{u}) \wedge \Pi_1 \vdash \Sigma_3\theta * \mathsf{P}(\vec{v})\theta * \Sigma \wedge \Pi_3\theta \wedge \Pi_1 \vdash F_4\theta * \Sigma \wedge \Pi_1$. Therefore, if the sub-goal entailment $F_4\theta * \Sigma \wedge \Pi_1 \vdash F_2$ can be proved, then the goal entailment holds. Here, we propagate the pure condition Π_1 through the chain as we want the antecedent of the sub-goal entailment to be the strongest in order to prove F_2.

$$\mathsf{LM_L}\ \frac{\mathcal{H},\ \mathcal{L} \cup \{\Sigma_3 \wedge \Pi_3 \vdash F_4\},\ F_4\theta * \Sigma \wedge \Pi_1 \vdash F_2}{\mathcal{H},\ \mathcal{L} \cup \{\Sigma_3 \wedge \Pi_3 \vdash F_4\},\ \Sigma_1 \wedge \Pi_1 \vdash F_2}\ \exists\theta, \Sigma.\ (\Sigma_1 \cong \Sigma_3\theta * \Sigma) \wedge (\Pi_1 \rightarrow \Pi_3\theta)$$

$$\mathsf{LM_R}\ \frac{\mathcal{H},\ \mathcal{L} \cup \{F_3 \vdash \exists\vec{w}.(\Sigma_4 \wedge \Pi_4)\},\ F_1 \vdash \exists\vec{x}.(F_3\theta * \Sigma \wedge \Pi_2)}{\mathcal{H},\ \mathcal{L} \cup \{F_3 \vdash \exists\vec{w}.(\Sigma_4 \wedge \Pi_4)\},\ F_1 \vdash \exists\vec{x}.(\Sigma_2 \wedge \Pi_2)}\ \exists\theta, \Sigma.\ (\Sigma_4\theta * \Sigma \cong \Sigma_2) \wedge (\vec{w}\theta \subseteq \vec{x})$$

Fig. 8. Lemma application rules

Lemma Application Rules. The two rules $\mathsf{LM_L}$, $\mathsf{LM_R}$ in Figure 8 derive a new sub-goal entailment by applying a lemma on the goal entailment's antecedent or consequent. In particular:

- The rule $\mathsf{LM_L}$ applies a lemma on the goal entailment's antecedent. It is similar to the induction application rule IH, except that we do not need to check the well-founded condition of the structural induction proof, since the applied lemma is already proved valid.
- The rule $\mathsf{LM_R}$ applies a lemma on the goal entailment's consequent. It also needs to perform an unification step: finding a substitution θ and a formula Σ so that the heap parts of the goal entailment's and the lemma's consequents are unified, i.e., $\Sigma_4\theta * \Sigma \cong \Sigma_2$. If this step succeeds, then $E_1 \triangleq \Sigma_4\theta * \Sigma \wedge \Pi_2 \vdash \Sigma_2 \wedge \Pi_2$ is valid. The lemma $F_3 \vdash \exists\vec{w}.(\Sigma_4 \wedge \Pi_4)$ implies that $F_3 \vdash \exists\vec{w}.\Sigma_4$ is valid. Hence, $F_3\theta \vdash \exists\vec{w}\theta.\Sigma_4\theta$ is valid by Theorem 3.8, following that $F_3\theta * \Sigma \wedge \Pi_2 \vdash \exists\vec{w}\theta.(\Sigma_4\theta * \Sigma \wedge \Pi_2)$ is also valid. By the rule's side condition $\vec{w}\theta \subseteq \vec{x}$, we have another valid entailment $E_2 \triangleq F_3\theta * \Sigma \wedge \Pi_2 \vdash \exists\vec{x}.(\Sigma_4\theta * \Sigma \wedge \Pi_2)$. Therefore, by proving the entailment $F_1 \vdash \exists\vec{x}.(F_3\theta * \Sigma \wedge \Pi_2)$ in this rule's premise and applying Theorem 3.6 twice sequentially on E_2 and then E_1, we can conclude that the goal entailment $F_1 \vdash \exists\vec{x}.(\Sigma_2 \wedge \Pi_2)$ is also valid.

4.2 Synthesis Rules for Unknown Entailments

Figure 9 presents synthesis rules, which deal with unknown entailments introduced in the lemma synthesis. These rules share the similar structure to the inference rules, except that each of them also contains a special premise indicating an assumption set $\mathcal{A}$ of the unknown relations. We will describe in details the synthesis rules as follows:

- *Axiom synthesis rules* U^1_Π, U^2_Π, U^1_Σ, U^2_Σ. These rules conclude the validity of their unknown goal entailments under the assumption sets $\mathcal{A}$ in the rules' premises. The rule U^1_Π and U^2_Π handle pure entailments with unknown relations either in the antecedents or the consequents. The rule U^1_Σ and U^2_Σ deal with unknown entailments whose antecedents contain non-empty spatial formulas while the consequents are pure formulas or vice versa. In both cases, the antecedents must be inconsistent to make the goal entailments valid. Here, we only create assumptions on the pure parts, i.e., $\Pi_1 \wedge U(\vec{x}) \rightarrow \mathit{false}$, since inconsistency in their heap parts, if any, can be detected earlier by the rule $\bot^1_\mathsf{L}$. Also, we do not consider the case that the unknown relations only appear in the consequents since these relations cannot make the antecedents inconsistent.
- *Induction hypothesis synthesis rule* U_{IH}. This rule applies an induction hypothesis to prove a derived unknown entailment. The induction hypothesis also contains unknown relations since it is recorded earlier from an unknown goal entailment. The rule U_{IH} is similar to the normal induction hypothesis application rule IH, except that it does not contain a side condition like $\Pi_1 \wedge U(\vec{x}) \rightarrow (\Pi_3 \wedge U(\vec{y}))\theta$, due to the appearance of the unknown relation U. Instead, this condition will be registered in the unknown assumption set $\mathcal{A}$ in the premises of U_{IH}.

$$\mathsf{U}^1_\Pi \ \frac{\mathcal{A} \triangleq \{\Pi_1 \wedge U(\vec{x}) \rightarrow \Pi_2\}}{\mathcal{H},\ \mathcal{L},\ \Pi_1 \wedge U(\vec{x}) \vdash \Pi_2} \qquad \mathsf{U}^1_\Sigma \ \frac{\mathcal{A} \triangleq \{\Pi_1 \wedge U(\vec{x}) \rightarrow \mathit{false}\}}{\mathcal{H},\ \mathcal{L},\ \Sigma_1 \wedge \Pi_1 \wedge U(\vec{x}) \vdash \Pi_2}\ \Sigma_1 \neq \mathsf{emp}$$

$$\mathsf{U}^2_\Pi \ \frac{\mathcal{A} \triangleq \{\Pi_1 \rightarrow \exists \vec{w}.(\Pi_2 \wedge U(\vec{x}))\}}{\mathcal{H},\ \mathcal{L},\ \Pi_1 \vdash \exists \vec{w}.(\Pi_2 \wedge U(\vec{x}))} \qquad \mathsf{U}^2_\Sigma \ \frac{\mathcal{A} \triangleq \{\Pi_1 \wedge U(\vec{x}) \rightarrow \mathit{false}\}}{\mathcal{H},\ \mathcal{L},\ \Pi_1 \wedge U(\vec{x}) \vdash \exists \vec{w}.(\Sigma_2 \wedge \Pi_2)}\ \Sigma_2 \neq \mathsf{emp}$$

$$\mathsf{U}_{\mathsf{IH}} \ \frac{\mathcal{H} \cup \{\Sigma_3 * \mathsf{P}(\vec{v}) \wedge \Pi_3 \wedge U(\vec{y}) \vdash F_4\},\ \mathcal{L},\ F_4\theta * \Sigma \wedge \Pi_1 \vdash F_2 \qquad \mathcal{A} \triangleq \{\Pi_1 \wedge U(\vec{x}) \rightarrow (\Pi_3 \wedge U(\vec{y}))\theta\}}{\mathcal{H} \cup \{\Sigma_3 * \mathsf{P}(\vec{v}) \wedge \Pi_3 \wedge U(\vec{y}) \vdash F_4\},\ \mathcal{L},\ \Sigma_1 * \mathsf{P}(\vec{u}) \wedge \Pi_1 \wedge U(\vec{x}) \vdash F_2}\ \dagger$$

with $\dagger:\ \mathsf{P}(\vec{u}) \prec \mathsf{P}(\vec{v});\ \exists \theta, \Sigma.\ (\Sigma_1 * \mathsf{P}(\vec{u}) \cong \Sigma_3\theta * \mathsf{P}(\vec{v})\theta * \Sigma)$

Fig. 9. Synthesis rules

4.3 The Proof Search Procedure

Figure 10 presents the core proof search procedure Prove of our proof system. Its first three inputs include an induction hypothesis set $\mathcal{H}$, a valid lemma set $\mathcal{L}$, and a goal entailment $F_1 \vdash F_2$, which correlate to an inference step of the proof system. We also use an additional input *mode* to control when new lemmas can be synthesized (if *mode* = SynLM) or strictly not (if *mode* = NoSyn). The procedure Prove returns the validity of the goal entailment, a set of new lemmas synthesized during the proof, and a set of assumptions to make the entailment valid.

There are two main contexts where this procedure is initially invoked:

- In the *entailment proving* phase, Prove is invoked to prove a *normal* entailment $F_1 \vdash F_2$, which does not contain any unknown relation, with the initial setting $\mathcal{H} = \varnothing, \mathcal{L} = \varnothing$, *mode* = SynLM.
- In the *lemma synthesis* phase, Prove is invoked either to prove an *unknown* entailment $F_1 \vdash F_2$ related to a lemma template, or to verify whether a discovered lemma is an inductive lemma. In the first case, Prove will establish sufficient assumptions about unknown relations appearing in the entailment so that the entailment can become valid. Moreover, Prove is invoked with *mode* = NoSyn in both the two cases to avoid entering into nested lemma synthesis phases.

In Figure 10, we also present formal specifications of Prove in pairs of pre- and post-conditions (*Requires/Ensures*). These specifications relate to three cases when Prove is invoked to prove an *unknown entailment* with the lemma synthesis always being disabled (*mode* = NoSyn), or to prove a *normal entailment* with the lemma synthesis being disabled (*mode* = NoSyn) or enabled (*mode* = SynLM). We write *res* to represent the returned result of Prove. In addition, $hasUnk(E)$

indicates that the entailment E has an unknown relation, $valid(\mathcal{L})$ and $valid(\mathcal{A})$ mean that all lemmas in $\mathcal{L}$ and all assumptions in $\mathcal{A}$ are *semantically valid*. Moreover, $valid_{\mathsf{ID}}(\mathcal{H}, \mathcal{L}, E)$ specifies that the entailment E is *semantically valid* under the induction hypothesis set $\mathcal{H}$ and the lemma set $\mathcal{L}$. We will refer to these specifications when proving the proof system's soundness. The formal verification of Prove w.r.t. these specifications is illustrated in the technical report [Ta et al. 2017].

Procedure Prove($\mathcal{H}, \mathcal{L}, F_1 \vdash F_2, mode$)

Input: $F_1 \vdash F_2$ is the goal entailment, $\mathcal{H}$ is a set of induction hypotheses, $\mathcal{L}$ is a set of valid lemmas, and *mode* controls whether new lemmas will be synthesized (SynLM) or strictly not (NoSyn)

Output: The goal entailment's validity (Valid$\langle\xi\rangle$ or Unkn, where ξ is the witness valid proof tree), a set of new synthesized lemmas, and a set of unknown assumptions

Requires: $(mode = \text{NoSyn}) \wedge hasUnk(F_1 \vdash F_2) \wedge valid(\mathcal{L})$
Ensures: $(res = (\text{Unkn}, \varnothing, \varnothing)) \ \vee \ \exists \xi, \mathcal{A}.((res = (\text{Valid}\langle\xi\rangle, \varnothing, \mathcal{A})) \wedge (valid(\mathcal{A}) \rightarrow valid_{\mathsf{ID}}(\mathcal{H}, \mathcal{L}, F_1 \vdash F_2)))$

Requires: $(mode = \text{NoSyn}) \wedge \neg hasUnk(F_1 \vdash F_2) \wedge valid(\mathcal{L})$
Ensures: $(res = (\text{Unkn}, \varnothing, \varnothing)) \ \vee \ \exists \xi.((res = (\text{Valid}\langle\xi\rangle, \varnothing, \varnothing)) \wedge valid_{\mathsf{ID}}(\mathcal{H}, \mathcal{L}, F_1 \vdash F_2))$

Requires: $(mode = \text{SynLM}) \wedge \neg hasUnk(F_1 \vdash F_2) \wedge valid(\mathcal{L})$
Ensures: $(res = (\text{Unkn}, \varnothing, \varnothing)) \vee \exists \xi, \mathcal{L}_{\text{syn}}.((res = (\text{Valid}\langle\xi\rangle, \mathcal{L}_{\text{syn}}, \varnothing)) \wedge valid(\mathcal{L}_{\text{syn}}) \wedge valid_{\mathsf{ID}}(\mathcal{H}, \mathcal{L}, F_1 \vdash F_2))$

1: $\overline{\mathcal{R}} \leftarrow \{\text{Unify}(\mathsf{R}, (\mathcal{H}, \mathcal{L}, F_1 \vdash F_2)) \mid \mathsf{R} \in \mathcal{R}\}$ *//Find applicable inference and synthesis rules from $\mathcal{R}$*
2: $\mathcal{L}_{\text{syn}} \leftarrow \varnothing$ *//Initialize the new synthesized lemma set*
3: **if** $(mode = \text{SynLM})$ **and** NeedLemmas$(F_1 \vdash F_2, \overline{\mathcal{R}})$ **then**
4: $\quad \mathcal{L}_{\text{syn}} \leftarrow \text{SynthesizeLemma}(\mathcal{L}, F_1 \vdash F_2)$ *//Synthesize new lemmas*
5: $\quad \overline{\mathcal{R}} \leftarrow \overline{\mathcal{R}} \cup \{\text{Unify}(\mathsf{R}, (\mathcal{H}, \mathcal{L}_{\text{syn}}, F_1 \vdash F_2)) \mid \mathsf{R} \in \{\text{LM}_\text{L}, \text{LM}_\text{R}\}\}$ *//Update lemma application rules*
6: **for each** $\overline{\mathsf{R}}$ **in** $\overline{\mathcal{R}}$ **do**
7: $\quad \mathcal{A} \leftarrow \varnothing$ *//Initialize the assumption set*
8: $\quad$ **if** IsSynthesisRule($\overline{\mathsf{R}}$) **then** $\mathcal{A} \leftarrow \text{Assumptions}(\overline{\mathsf{R}})$ *//$\overline{\mathsf{R}} \in \{\cup^1_\Pi, \cup^2_\Pi, \cup^1_\Sigma, \cup^2_\Sigma, \cup_{IH}\}$*
9: $\quad$ **if** IsAxiomRule($\overline{\mathsf{R}}$) **then** *//$\overline{\mathsf{R}} \in \{\vdash_\Pi, \bot^1_L, \bot^2_L, \cup^1_\Pi, \cup^2_\Pi, \cup^1_\Sigma, \cup^2_\Sigma\}$*
10: $\quad\quad \xi \leftarrow \text{CreateWitnessProofTree}(F_1 \vdash F_2, \overline{\mathsf{R}}, \varnothing)$
11: $\quad\quad$ **return** $(\text{Valid}\langle\xi\rangle, \mathcal{L}_{\text{syn}}, \mathcal{A})$
12: $\quad (\mathcal{H}_i, \mathcal{L}_i, F_{1i} \vdash F_{2i})_{i=1\ldots n} \leftarrow \text{Premises}(\overline{\mathsf{R}})$
13: $\quad (r_i, \mathcal{L}^i_{\text{syn}}, \mathcal{A}_i)_{i=1\ldots n} \leftarrow \text{Prove}(\mathcal{H}_i, \ \mathcal{L} \cup \mathcal{L}_{\text{syn}} \cup \mathcal{L}_i, \ F_{1i} \vdash F_{2i}, mode)_{i=1\ldots n}$ *//Prove all sub-goals*
14: $\quad$ **if** $r_i = \text{Valid}\langle\xi_i\rangle$ **for all** $i = 1 \ldots n$ **then**
15: $\quad\quad \xi \leftarrow \text{CreateWitnessProofTree}(F_1 \vdash F_2, \overline{\mathsf{R}}, \{\xi_1, \ldots, \xi_n\})$
16: $\quad\quad$ **return** $(\text{Valid}\langle\xi\rangle, \ \mathcal{L}_{\text{syn}} \cup \mathcal{L}^1_{\text{syn}} \cup \ldots \cup \mathcal{L}^n_{\text{syn}}, \ \mathcal{A} \cup \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_n)$
17: **return** $(\text{Unkn}, \varnothing, \varnothing)$ *//All rules fail to prove $F_1 \vdash F_2$*

Fig. 10. The main proof search procedure with description (Input/Output) and specification (Requires/Ensures)

Given the goal entailment $F_1 \vdash F_2$, the procedure Prove first finds from all rules $\mathcal{R}$ (Figures 6, 7, 8, and 9) a set of potential inference and synthesis rules $\overline{\mathcal{R}}$, which can be unified with $F_1 \vdash F_2$ (line 1). When the lemma synthesis mode is enabled ($mode$ = SynLM), it invokes a subroutine NeedLemmas to examine the selected rules $\overline{\mathcal{R}}$ and the goal entailment $F_1 \vdash F_2$ to decide whether it really needs to synthesize new lemmas (line 3). Note that the input valid lemma set $\mathcal{L}$ is also exploited to make the lemma synthesis more effective (line 4). The new synthesized lemmas, if any, will be utilized to discover new lemma application rules (line 5). Thereafter, Prove successively applies each rule $\overline{\mathsf{R}} \in \overline{\mathcal{R}}$ to derive new sub-goal entailments, as in the premises of $\overline{\mathsf{R}}$, and recursively searches for their proofs (lines 6 – 16). It returns the *valid* result (Valid$\langle\xi\rangle$, where ξ is the witness proof tree) if the selected rule $\overline{\mathsf{R}}$ does not introduce any new sub-goals (lines 9 – 11), or all derived

sub-goals are successfully proved (lines 12 – 16). In essence, the proof tree ξ is composed of a root (the goal entailment $F_1 \vdash F_2$), a label (the applied rule $\bar{\mathsf{R}}$), and sub-trees, if any, corresponding to the sub-goal entailments' proofs (lines 10, 15). Its form is intuitively similar to the proof trees depicted in Figures 1, 2, and 3. On the other hand, Prove announces the *unknown* result (Unkn) when all selected rules in $\overline{\mathcal{R}}$ fail to prove the goal entailment (line 17). Besides, Prove also returns a set of new synthesized lemmas and a set of unknown assumptions. These lemmas are discovered when Prove is invoked in the *entailment proving* phase. The unknown assumptions are collected by the synthesis rules (line 8), when Prove is executed in the *lemma synthesis* phase.

Details of the lemma synthesis will be presented in Section 5. In the following, we will explain when Prove decides to synthesize new lemmas (line 3). The procedure NeedLemmas returns *true* when all the following conditions are satisfied:

- $F_1 \vdash F_2$ *is not* an unknown entailment, which implies that lemmas are possibly needed.
- $\overline{\mathcal{R}}$ *does not contain* any axiom or normalization ($\vdash_\Pi$, $\bot^1_\mathsf{L}$, $\bot^2_\mathsf{L}$, $\exists_\mathsf{L}$, $\exists_\mathsf{R}$, $=_\mathsf{L}$, E_L, E_R), matching rules of identical heap predicates ($*\mapsto$, $*\mathsf{P}$), or unfolding rules that introduce identical predicates (ID, P_R). This condition indicates that all rules in $\overline{\mathcal{R}}$ cannot make any immediate proof progress.
- $\overline{\mathcal{R}}$ *does not have* any induction hypothesis or lemma application rules (IH, LM_L, LM_R), or any case analysis rule (CA) that potentially leads to the application of an induction hypothesis or a lemma. This condition conveys that existing induction hypotheses and lemmas are inapplicable
- $F_1 \vdash F_2$ *is not* a good induction hypothesis candidate, which indicates that the induction hypothesis recorded from $F_1 \vdash F_2$ cannot be used to prove other derived entailments.

While the first three conditions can be checked syntactically on $F_1 \vdash F_2$ and $\overline{\mathcal{R}}$, the last condition can be tested by a trial and error method. Specifically, $F_1 \vdash F_2$ will firstly be recorded as a temporary induction hypothesis candidate, and then each inductive heap predicate in F_1 will be consecutively unfolded to search for an application of the recorded induction hypothesis via the rule IH. If no induction hypothesis application can be found, then $F_1 \vdash F_2$ is evidently not a good candidate.

4.4 Soundness of the Proof System

Recall that our proof search procedure Prove is implemented in a recursive manner. When it is invoked to prove a normal goal entailment E, which does not contain any unknown relation, the initial setting $\mathcal{H} = \varnothing$, $\mathcal{L} = \varnothing$, $mode = \textsc{SynLM}$ indicates that no induction hypothesis or lemma is provided beforehand, and the proof system can synthesize new lemmas to assist in proving E. When synthesizing the new supporting lemmas, the proof system can be utilized to prove an *unknown entailment* related to a lemma template or to verify a discovered lemma, which is a *normal entailment* not containing any unknown relation. In addition, the proof system is also invoked to prove sub-goal entailments, which are *normal entailments* derived from E. All of these scenarios are summarized by the three specifications in Figure 10.

In the following, we present Propositions 4.1, 4.2, and 4.3, which respectively specify the soundness of our proof system in the three typical scenarios: (i) proving an unknown entailment with the lemma-synthesis-disabled mode (the first pre/post specification of Prove), (ii) verifying a discovered lemma, i.e., proving a normal entailment with the lemma-synthesis-disabled mode (the second pre/post specification), or (iii) proving a normal entailment with the lemma-synthesis-enabled mode (the third pre/post specification). Finally, we describe the overall soundness of the proof system in Theorem 4.4 when Prove is invoked with the initial setting $\mathcal{H} = \varnothing$, $\mathcal{L} = \varnothing$, $mode = \textsc{SynLM}$. Note that Propositions 4.1 and 4.2 are relevant to the lemma synthesis in Section 5. Proposition 4.3 directly relates to the overall soundness in Theorem 4.4.

Proposition 4.1 (Proof of an unknown entailment). *Given an unknown entailment E. If the procedure* Prove *returns* Valid$\langle_\rangle$ *and generates an assumption set $\mathcal{A}$ when proving E in the lemma-synthesis-disabled mode (NoSyn), using an empty induction hypothesis set ($\mathcal{H}$=$\varnothing$) and a valid lemma set $\mathcal{L}$ as its inputs, then E is semantically valid, given that all assumptions in $\mathcal{A}$ are valid.*

Proposition 4.2 (Proof of a normal entailment when the lemma synthesis is disabled). *Given a normal entailment E which does not contain any unknown relation. If the procedure* Prove *returns* Valid$\langle_\rangle$ *when proving E in the lemma-synthesis-disabled mode (NoSyn), using an empty induction hypothesis set ($\mathcal{H}$=$\varnothing$) and a valid lemma set $\mathcal{L}$ as its inputs, then E is semantically valid.*

Proposition 4.3 (Proof of a normal entailment when the lemma synthesis is enabled). *Given a normal entailment E which does not contain any unknown relation. If the procedure* Prove *returns* Valid$\langle_\rangle$ *and synthesizes a set of lemmas $\mathcal{L}_{\text{syn}}$ when proving E in the lemma-synthesis-enabled mode (SynLM), using an empty induction hypothesis set ($\mathcal{H}$=$\varnothing$) and a valid lemma set $\mathcal{L}$ as its inputs, then the entailment E and all lemmas in $\mathcal{L}_{\text{syn}}$ are semantically valid.*

Proofs of Propositions 4.1, 4.2, 4.3. We first show that all inference and synthesis rules are sound, and the proof system in the NoSyn mode is sound. Based on that, we can prove Propositions 4.1 and 4.2. The proof of Proposition 4.3 will be argued based on the lemma synthesis's soundness in Section 5. Details of all these proofs are presented in the technical report [Ta et al. 2017]. □

Theorem 4.4 (The overall soundness of the proof system). *Given a normal entailment E which does not contain any unknown relation, if the procedure* Prove *returns* Valid$\langle_\rangle$ *when proving E in the initial context that there is no induction hypothesis or lemma provided beforehand and the lemma synthesis is enabled ($\mathcal{H}=\varnothing$, $\mathcal{L}=\varnothing$, mode = SynLM), then E is semantically valid.*

Proof. Since E does not contain any unknown relation, both the input induction hypothesis set and lemma set are empty ($\mathcal{H}$=$\varnothing$, $\mathcal{L}$=$\varnothing$), and Prove is invoked in the SynLM mode, it follows from Proposition 4.3 that if Prove returns Valid$\langle_\rangle$ when proving E, then E is semantically valid. □

5 THE LEMMA SYNTHESIS FRAMEWORK

We are now ready to describe our lemma synthesis framework. It consists of a main procedure, presented in Subsection 5.1, and auxiliary subroutines, described in Subsections 5.2, 5.3, and 5.4. Similar to the proof system in Section 4, we also provide the input/output description and the formal specification for each of these synthesis procedures. We use the same keyword *res* to represent the returned result and $valid(\mathcal{L})$ indicates that all lemmas in $\mathcal{L}$ are semantically valid.

5.1 The Lemma Synthesis Procedure

Figure 11 presents the main lemma synthesis procedure (SynthesizeLemma). Its inputs include a goal entailment $F_1 \vdash F_2$ which needs to be proved by new lemmas, and a set of previously synthesized lemmas $\mathcal{L}$ which will be exploited to make the current synthesis more effective. The procedure SynthesizeLemma first identifies a set of desired lemma templates based on the entailment's heap structure, via the invocation of the procedure CreateTemplate (line 18). These lemma

Procedure SynthesizeLemma($\mathcal{L}, F_1 \vdash F_2$)

Input: $F_1 \vdash F_2$ is the goal entailment, $\mathcal{L}$ is a valid lemma set
Output: A set of new synthesized lemmas
Requires: $valid(\mathcal{L})$
Ensures: $valid(res)$

18: **for each** $\Sigma_1 \vdash \exists\vec{v}.\Sigma_2$ **in** CreateTemplate($\mathcal{L}, F_1 \vdash F_2$) **do**
19: $\quad\mathcal{L}_1 \leftarrow$ RefineAnte($\mathcal{L}$, $\Sigma_1 \wedge true \vdash \Sigma_2$)
20: $\quad\mathcal{L}_2 \leftarrow$ RefineConseq($\mathcal{L}$, $\Sigma_1 \vdash \exists\vec{v}.\Sigma_2$)
21: $\quad$**if** $(\mathcal{L}_1 \cup \mathcal{L}_2) \neq \varnothing$ **then return** $(\mathcal{L}_1 \cup \mathcal{L}_2)$
22: **return** $\varnothing$

Fig. 11. The main lemma synthesis procedure

templates are of the form $\Sigma_1 \vdash \exists\vec{v}.\Sigma_2$, in which Σ_1 and Σ_2 are spatial formulas constructed from heap predicate symbols appearing in F_1 and F_2, respectively, and $\vec{v}$ are all free variables in Σ_2. By this construction, each synthesized lemma will share a similar heap structure with the goal entailment, hence they can be unified by the lemma application rules $\mathsf{LM_L}$ and $\mathsf{LM_R}$. We will formally define the lemma templates in Subsection 5.2. In our implementation, CreateTemplate returns a list of possible lemma templates, which are sorted in the ascending order of their simplicity, i.e., templates containing less spatial atoms are on the top of the list. Moreover, any template sharing the same heap structure with a previously synthesized lemma in $\mathcal{L}$ will not be considered.

The framework then successively refines each potential lemma template by continuously discovering and adding in *pure constraints* of its variables until valid *inductive lemmas* are found. In essence, for a given lemma template, the refinement is performed by a 3-step recipe:

(1) Establishing an unknown relation representing a desired constraint inside the template and creating an unknown entailment.
(2) Proving the unknown entailment by structural induction and collecting assumptions about the unknown relation.
(3) Solving the assumptions to find out the actual definition of the unknown relation, thus discovering the desired inductive lemma.

There are two possible places to refine a lemma template: on its antecedent (line 19) and its consequent (line 20). We aim to synthesize a lemma which has an *as weak as possible* antecedent or an *as strong as possible* consequent. We will elaborate the details in Subsections 5.3 and 5.4. Our framework *immediately* returns a non-empty set of synthesized lemmas (line 21) once it successfully refines a template. Otherwise, it returns an empty set ($\varnothing$) indicating that no lemma can be synthesized (line 22).

5.2 Discovering Lemma Templates

The lemma templates for a given goal entailment can be discovered from the entailment's heap structure. In the followings, we present the formal definition of the lemma templates and also illustrate by examples how to create them.

Definition 5.1 (Lemma template). A lemma template for a goal entailment $F_1 \vdash F_2$ is an entailment of the form $\Sigma_1 \vdash \exists\vec{v}.\Sigma_2$, where:

(1) Σ_1 and Σ_2 are spatial formulas containing at least one inductive heap predicate.
(2) Heap predicate symbols in Σ_1 and Σ_2 are sub-multisets of those in F_1 and F_2.
(3) Variables in Σ_1 and Σ_2 are separately named (no variables appear twice) and $\vec{v} \equiv \mathsf{FV}(\Sigma_2)$.

The condition (1) limits the templates for only inductive lemmas. The condition (2) guarantees that the synthesized lemmas are *unifiable* with the goal entailment via applications of the rules $\mathsf{LM_L}$ and $\mathsf{LM_R}$. Moreover, the condition (3) ensures that each template is as general as possible so that desirable pure constraints can be subsequently discovered in the next phases.

For instance, the following lemma templates can be created given the motivating entailment in Section 2: $E_1 \triangleq \mathsf{dllrev}(x,y,u,v,n) * \mathsf{dll}(v,u,z,t,200) \wedge n{\geq}100 \vdash \exists r.(\mathsf{dll}(x,y,r,z,n{+}199) * z{\mapsto}r,t)$. Note that the template T_1 is used to synthesize the lemma $L_1 \triangleq \mathsf{dllrev}(a,b,c,d,m) \vdash \mathsf{dll}(a,b,c,d,m)$.

$$T_1 \triangleq \mathsf{dllrev}(x_1,x_2,x_3,x_4,n_1) \vdash \exists x_5,x_6,x_7,x_8,n_2.\mathsf{dll}(x_5,x_6,x_7,x_8,n_2)$$
$$\mathsf{dll}(x_1,x_2,x_3,x_4,n_1) \vdash \exists x_5,x_6,x_7,x_8,x_9,x_{10},x_{11},n_2.(\mathsf{dll}(x_5,x_6,x_7,x_8,n_2) * x_9{\mapsto}x_{10},x_{11})$$
$$\mathsf{dllrev}(x_1,x_2,x_3,x_4,n_1) \vdash \exists x_5,x_6,x_7,x_8,x_9,x_{10},x_{11},n_2.(\mathsf{dll}(x_5,x_6,x_7,x_8,n_2) * x_9{\mapsto}x_{10},x_{11})$$
$$\mathsf{dllrev}(x_1,x_2,x_3,x_4,n_1) * \mathsf{dll}(x_5,x_6,x_7,x_8,n_2) \vdash \exists x_9,x_{10},x_{11},x_{12},n_3.\mathsf{dll}(x_9,x_{10},x_{11},x_{12},n_3)$$

Similarly, there can be several possible lemma templates relating to the entailments E_2 and E_3 in Figure 1. Among them, the templates T_2 and T_3 below are used to synthesize the lemmas L_2 and L_3:

$$T_2 \triangleq \mathsf{dll}(x_1, x_2, x_3, x_4, n_1) * \mathsf{dll}(x_5, x_6, x_7, x_8, n_2) \vdash \exists x_9, x_{10}, x_{11}, x_{12}, n_3.\mathsf{dll}(x_9, x_{10}, x_{11}, x_{12}, n_3)$$
$$T_3 \triangleq \mathsf{dll}(x_1, x_2, x_3, x_4, n_1) \vdash \exists x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, n_2.(\mathsf{dll}(x_5, x_6, x_7, x_8, n_2) * x_9 \mapsto x_{10}, x_{11})$$

5.3 Refining Lemma Templates' Antecedents

Figure 12 presents the antecedent refinement procedure (RefineAnte), which aims to strengthen the antecedent of a lemma template $\Sigma_1 \wedge \Pi_1 \vdash \Sigma_2$ with pure constraints of all its variables. Note that when RefineAnte is invoked in the first time by SynthesizeLemma (Figure 11, line 19), Π_1 is set to *true* and the existential quantification $\exists \vec{v}$ in the original template $\Sigma_1 \vdash \exists \vec{v}.\Sigma_2$ is removed since the antecedent will be strengthened with constraints of all variables in the template.

Procedure RefineAnte($\mathcal{L}$, $\Sigma_1 \wedge \Pi_1 \vdash \Sigma_2$)

Input: $\Sigma_1 \wedge \Pi_1 \vdash \Sigma_2$ is a lemma template, $\mathcal{L}$ is a valid lemma set
Output: a set of at most one synthesized lemma
Requires: $valid(\mathcal{L})$
Ensures: $(res = \varnothing) \vee \exists\, \Pi_1'.(res = \{\Sigma_1 \wedge \Pi_1 \wedge \Pi_1' \vdash \Sigma_2\} \wedge valid(res))$

23: $\vec{u} \leftarrow \mathsf{FreeVars}(\Sigma_1, \Sigma_2, \Pi_1)$
24: $U(\vec{u}) \leftarrow \mathsf{CreateUnknownRelation}(\vec{u})$
25: $r, \_, \mathcal{A} \leftarrow \mathsf{Prove}(\varnothing,\ \mathcal{L},\ \Sigma_1 \wedge \Pi_1 \wedge U(\vec{u}) \vdash \Sigma_2,\ \textsc{NoSyn})$
26: **if** $r = \textsc{Valid}\langle\_\rangle$ **then**
27: **for each** $\mathcal{A}'$ **in** $\mathsf{SuperSet}(\mathcal{A})$ **do**
28: $\overline{U}(\vec{u}) \leftarrow \mathsf{Solve}(\mathcal{A}', U(\vec{u}))$
29: **if** $\Sigma_1 \wedge \Pi_1 \wedge \overline{U}(\vec{u}) \not\equiv \mathit{false}$ **then**
30: $L \leftarrow (\Sigma_1 \wedge \Pi_1 \wedge \overline{U}(\vec{u}) \vdash \Sigma_2)$
31: $r', \_, \_ \leftarrow \mathsf{Prove}(\varnothing,\ \mathcal{L},\ L,\ \textsc{NoSyn})$ // *Verify ...*
32: **if** $r' = \textsc{Valid}\langle\xi\rangle$ **and** $\mathsf{IH} \in \xi$ **then** // *and return ...*
33: **return** $\{L\}$ // *the first inductive lemma,*
34: **else return** $\mathsf{RefineAnte}(\mathcal{L},\ L)$ // *or continue refining.*
35: **return** $\varnothing$ // *No lemmas are found.*

Fig. 12. Refining a lemma template's antecedent

Initially, RefineAnte creates an unknown entailment $\Sigma_1 \wedge \Pi_1 \wedge U(\vec{u}) \vdash \Sigma_2$, where $U(\vec{u})$ is an unknown relation of all variables $\vec{u}$ in $\Sigma_1, \Sigma_2, \Pi_1$ (lines 23, 24). Then, it proves the entailment by induction and collects a set $\mathcal{A}$ of unknown assumptions, if any, about $U(\vec{u})$ (line 25). In this proof derivation (also in other invocations of Prove in the lemma synthesis), we prevent the proof system from synthesizing new lemmas (by passing NoSyn) to avoid initiating a new lemma synthesis phase inside the current synthesis, i.e., to prohibit the nested lemma synthesis. In addition, the assumption set $\mathcal{A}$ will be examined, via the procedure Solve, to discover a solution of $U(\vec{u})$ (lines 27, 28). We implement in Solve a constraint solving technique using Farkas' lemma [Colón et al. 2003; Schrijver 1986]. This technique assigns $U(\vec{u})$ to a predefined linear arithmetic formula with unknown coefficients, and applies Farkas' lemma to transform $\mathcal{A}$ into a set of constraints involving only the unknown coefficients. Then, it utilizes an off-the-shelf prover such as Z3 [Moura and Bjørner 2008] to find a concrete model of the coefficients, thus obtains the actual definition $\overline{U}(\vec{u})$ of $U(\vec{u})$. We will describe this technique in Subsection 5.6.

Furthermore, we aim to find a *non-spurious* solution $\overline{U}(\vec{u})$ which does not refute the antecedent, i.e., $\Sigma_1 \wedge \Pi_1 \wedge \overline{U}(\vec{u}) \not\equiv \mathit{false}$, to avoid creating an *useless* lemma: $\mathit{false} \vdash \Sigma_2$ (line 29). To examine this refutation, we follow the literature [Brotherston et al. 2014; Le et al. 2016] to implement an unsatisfiability checking algorithm, which over-approximates a symbolic-heap formula to a pure formula and invokes the off-the-shelf prover to check the pure formula's unsatisfiability, thus concludes about the unsatisfiability of the original formula.

In general, discovering a *non-spurious* solution $\overline{U}(\vec{u})$ is challenging, because:

– The assumption set $\mathcal{A}$ can be complicated since the parameters $\vec{u}$ of the unknown relation $U(\vec{u})$ are all variables in the lemma template. This complexity can easily overwhelm the underlying prover when finding the model of the corresponding unknown coefficients.

– The discovered proof tree of the unknown entailment might not be similar to the actual proof tree of the desired lemma, due to the occurrence of the unknown relation. Therefore, the set $\mathcal{A}$ might contain *noise* assumptions of $U(\vec{u})$, which results in a spurious solution. Nonetheless, a part of the expected solution can still be discovered from a subset of $\mathcal{A}$, which corresponds to the common part of the discovered and the desired proof trees.

The above challenges inspire us to design an *exhaustive approach* to solve $\mathcal{A}$ (line 27). In particular, RefineAnte first solves the entire set $\mathcal{A}$ (the first element in SuperSet($\mathcal{A}$)) to find a *complete* solution, which satisfies *all* assumptions in $\mathcal{A}$. If such solution is *not* available, RefineAnte iteratively examines each subset of $\mathcal{A}$ (among the remaining elements in SuperSet($\mathcal{A}$)) to discover a *partial* solution, which satisfies *some* assumptions in $\mathcal{A}$.

The discovered solution (complete or partial) will be verified whether it can form a valid inductive lemma. In particular, the proof system will be invoked to prove the candidate lemma L (line 31). The verification is successful when L is proved valid and its witness proof tree ξ contains an induction hypothesis application (labeled by IH) (line 32). The latter condition ensures that the returned lemma is actually an inductive lemma.

We also follow an *incremental approach* to refine the lemma template's antecedent. That is, the antecedent will be strengthened with the discovered solution $\overline{U}(\vec{u})$ to derive a new template. The new template will be refined again until the first valid lemma is discovered (lines 32 – 34). Note that this refinement stops at the first solution to ensure that the discovered antecedent is as weak as possible. Finally, RefineAnte returns $\varnothing$ if no valid inductive lemma can be discovered (line 35).

For example, given the template $T_1 \triangleq \mathsf{dllrev}(x_1,x_2,x_3,x_4,n_1) \vdash \exists x_5,x_6,x_7,x_8,n_2.\mathsf{dll}(x_5,x_6,x_7,x_8,n_2)$, RefineAnte creates an unknown relation $U(x_1,\ldots,x_8,n_1,n_2)$ and introduces the entailment E_{u_1}:

$$E_{u_1} \triangleq \mathsf{dllrev}(x_1,x_2,x_3,x_4,n_1) \wedge U(x_1,\ldots,x_8,n_1,n_2) \vdash \mathsf{dll}(x_5,x_6,x_7,x_8,n_2)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{A}_1 \triangleq \{x_1{=}x_3 \wedge n_1{=}1 \wedge U(\vec{x},\vec{n}) \rightarrow x_1{=}x_5 \wedge x_2{=}x_6 \wedge x_4{=}x_8 \wedge x_5{=}x_7 \wedge n_2{=}1\}}{x_1{=}x_3 \wedge n_1{=}1 \wedge U(\vec{x},\vec{n}) \vdash x_1{=}x_5 \wedge x_2{=}x_6 \wedge x_4{=}x_8 \wedge x_5{=}x_7 \wedge n_2{=}1}\,\mathsf{U}^1_{\Pi}}{x_1{\mapsto}x_2, x_4 \wedge x_1{=}x_3 \wedge n_1{=}1 \wedge U(\vec{x},\vec{n}) \vdash x_5{\mapsto}x_6, x_8 \wedge x_5{=}x_7 \wedge n_2{=}1}\,{*}{\mapsto}}{x_1{\mapsto}x_2, x_4 \wedge x_1{=}x_3 \wedge n_1{=}1 \wedge U(\vec{x},\vec{n}) \vdash \mathsf{dll}(x_5,x_6,x_7,x_8,n_2)}\,\mathsf{P_R} \quad \dfrac{\dfrac{\dfrac{\mathcal{A}_2 \triangleq \{U(\vec{x},\vec{n}) \rightarrow \mathit{false}\}}{x_3{\mapsto}u, x_4 \wedge U(\vec{x},\vec{n}) \vdash a_5{=}x_5 \wedge a_6{=}x_6 \wedge a_7{=}x_7 \wedge a_8{=}x_8 \wedge b_2{=}n_2}\,\mathsf{U}^1_{\Sigma}}{\mathsf{dll}(a_5,a_6,a_7,a_8,b_2) * x_3{\mapsto}u, x_4 \wedge U(\vec{x},\vec{n}) \vdash \mathsf{dll}(x_5,x_6,x_7,x_8,n_2)}\,{*}\mathsf{P} \quad \mathcal{A}_3 \triangleq \{U(\vec{x},\vec{n}) \rightarrow U(\vec{a},\vec{b})\theta\}}{\mathsf{dllrev}(x_1,x_2,u,x_3,n_1{-}1) * x_3{\mapsto}u, x_4 \wedge U(\vec{x},\vec{n}) \vdash \mathsf{dll}(x_5,x_6,x_7,x_8,n_2)}\,\mathsf{U_{IH}}}{E_{u_1} \triangleq \mathsf{dllrev}(x_1,x_2,x_3,x_4,n_1) \wedge U(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,n_1,n_2) \vdash \mathsf{dll}(x_5,x_6,x_7,x_8,n_2)}\,\mathsf{ID}$$

Fig. 13. A possible proof tree of the unknown entailment E_{u_1}

where $U(\vec{x},\vec{n}) \equiv U(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,n_1,n_2)$; $U(\vec{a},\vec{b}) \equiv U(a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,b_1,b_2)$;
the rule ID is performed to record the IH $H \triangleq \mathsf{dllrev}(a_1,a_2,a_3,a_4,b_1) \wedge U(\vec{a},\vec{b}) \vdash \mathsf{dll}(a_5,a_6,a_7,a_8,b_2)$;
the rule $\mathsf{U_{IH}}$ applies H with $\theta = [x_1/a_1, x_2/a_2, u/a_3, x_3/a_4, n_1{-}1/b_1]$;

Figure 13 presents a possible proof tree of E_{u_1}. From this proof, we obtain a set $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3$ of three unknown assumptions about the relation $U(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,n_1,n_2)$:

(1) $x_1{=}x_3 \wedge n_1{=}1 \wedge U(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,n_1,n_2) \rightarrow x_1{=}x_5 \wedge x_2{=}x_6 \wedge x_4{=}x_8 \wedge x_5{=}x_7 \wedge n_2{=}1$
(2) $U(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,n_1,n_2) \rightarrow \mathit{false}$
(3) $U(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,n_1,n_2) \rightarrow U(x_1,x_2,u,x_3,a_5,a_6,a_7,a_8,n_1{-}1,b_2)$

Our framework first attempts to solve the full assumption set $\mathcal{A}$. Unfortunately, there is only a *spurious solution* $\overline{U}(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,n_1,n_2) \equiv \mathit{false}$, since the assumption (2) is too strong.

It then tries to find another solution by *partially solving* the set $\mathcal{A}$. In this case, it can discover the following partial solution $\overline{U}$ when solving a subset containing only the assumption (1):

$$\overline{U}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, n_1, n_2) \equiv x_1{=}x_5 \wedge x_2{=}x_6 \wedge x_3{=}x_7 \wedge x_4{=}x_8 \wedge n_1{=}n_2$$

Since the above is only a partial solution, the framework needs to verify whether it can construct a *valid inductive lemma*. Indeed, $\overline{U}$ can be used to derive the following entailment $\overline{E}_{u_1}$, which can be simplified to become $\mathsf{dllrev}(x_1, x_2, x_3, x_4, n_1) \vdash \mathsf{dll}(x_1, x_2, x_3, x_4, n_1)$. The latter entailment can be equivalently transformed into the motivating lemma L_1 by a renaming on its variables.

$$\overline{E}_{u_1} \triangleq \mathsf{dllrev}(x_1, x_2, x_3, x_4, n_1) \wedge x_1{=}x_5 \wedge x_2{=}x_6 \wedge x_3{=}x_7 \wedge x_4{=}x_8 \wedge n_1{=}n_2 \vdash \mathsf{dll}(x_5, x_6, x_7, x_8, n_2)$$

5.4 Refining Lemma Templates' Consequents

The consequent refinement is presented in the procedure RefineConseq (Figure 14). Unlike the antecedent refinement, this refinement is not straightforward by simply adding pure constraints into the template's consequent, since the existing template's antecedent might not be strong enough to prove any formulas derived from the consequent. For example, all entailments derived from adding pure constraints to only the consequent of the lemma template T_3 are invalid, because when $n_1{=}1$, the list $\mathsf{dll}(x_1, x_2, x_3, x_4, n_1)$ has the length of 1, thus cannot be split into a singleton heap $x_9{\mapsto}x_{10}, x_{11}$ and a list $\mathsf{dll}(x_5, x_6, x_7, x_8, n_2)$, whose length is at least 1, by the definition of dll.

$$T_3 \triangleq \mathsf{dll}(x_1, x_2, x_3, x_4, n_1) \vdash \exists x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, n_2.(\mathsf{dll}(x_5, x_6, x_7, x_8, n_2) * x_9{\mapsto}x_{10}, x_{11})$$

To overcome this problem, we decompose the consequent refinement into two phases, *preprocessing* and *fine-tuning*. For an input lemma template $\Sigma_1 \vdash \exists\vec{v}.\Sigma_2$, in the first phase, the framework infers a pure constraint Π_1 so that the entailment $\Sigma_1 \wedge \Pi_1 \vdash \exists\vec{v}.\Sigma_2$ is valid (lines 36, 37). In the second phase, it incrementally strengthens the consequent of the derived entailment until discovers a valid inductive lemma (line 38). Note that, we retain the existential quantification in this consequent, and any constraints added into the consequent will be bound by this quantification. These two phases will be elaborated in details as follows.

Procedure RefineConseq($\mathcal{L}$, $\Sigma_1 \vdash \exists\vec{v}.\Sigma_2$)

Input: $\Sigma_1 \vdash \exists\vec{v}.\Sigma_2$ is a lemma template, $\mathcal{L}$ is a set of valid lemmas
Output: a set of at most one synthesized lemma
Requires: $valid(\mathcal{L})$
Ensures: $(res = \varnothing) \vee \exists\,\Pi_1,\Pi_2.(res = \{\Sigma_1 \wedge \Pi_1 \vdash \exists\vec{v}.(\Sigma_2 \wedge \Pi_2)\} \wedge valid(res))$

36: $\mathcal{T} \leftarrow$ Preprocess($\mathcal{L}$, $\Sigma_1 \vdash \exists\vec{v}.\Sigma_2$)
37: **if** $\mathcal{T} = \{\Sigma_1 \wedge \Pi_1 \vdash \exists\vec{v}.\Sigma_2\}$ **then**
38: **return** FineTuneConseq($\mathcal{L}$, $\Sigma_1 \wedge \Pi_1 \vdash \exists\vec{v}.(\Sigma_2 \wedge true)$)
39: **return** $\varnothing$

Fig. 14. Refining a lemma template's consequent

5.4.1 Preprocessing Lemma Templates. Figure 15 presents our antecedent preprocessing procedure (Preprocess). Similar to the antecedent refinement in Section 5.3, this procedure strengthens the antecedent of the template $\Sigma_1 \vdash \exists\vec{v}.\Sigma_2$ with a *non-spurious* condition Π_1 to make the lemma template valid. We also prevent the framework from entering nested lemma synthesis phases by invoking Prove in the NoSyn mode (line 43).

However, this preprocessing step differs from the antecedent refinement (Section 5.3) when it creates an unknown entailment by introducing the unknown relation $U(\vec{u})$ on only the antecedent's free variables $\vec{u}$ (lines 40, 41). We also equip the consequent $\exists\vec{v}.\Sigma_2$ of the unknown entailment with a conjunction Π_{inv} of its inductive heap predicates' pure invariants (lines 42, 43). These invariants are pure formulas representing Boolean constraints of the predicates' variables. We will briefly describe the invariant construction in Section 5.5. This construction is also well-studied in separation logic literature [Brotherston et al. 2014; Chin et al. 2012; Le et al. 2016].

In theory, equipping additional pure invariants of inductive heap predicates *does not* weaken or strengthen the lemma template's consequent, i.e., $\Sigma_2 \equiv \Sigma_2 \wedge \Pi_{inv}$. In our approach, Preprocess solves the entire assumption constraint set $\mathcal{A}$ at once (line 45), and not incrementally as in the antecedent refinement (Section 5.3). Therefore, the additional pure invariant Π_{inv} is useful for Preprocess to solve the entire set $\mathcal{A}$ more precisely and effectively.

Procedure Preprocess($\mathcal{L}$, $\Sigma_1 \vdash \exists\vec{v}.\Sigma_2$)

Input: $\Sigma_1 \vdash \exists\vec{v}.\Sigma_2$ is a lemma template, $\mathcal{L}$ is a valid lemma set
Output: a set of at most one refined template

Requires: $valid(\mathcal{L})$
Ensures: $(res = \varnothing) \vee \exists\,\Pi_1.(res = \{\Sigma_1 \wedge \Pi_1 \vdash \exists\vec{v}.\Sigma_2\} \wedge valid(res))$

40: $\vec{u} \leftarrow$ FreeVars(Σ_1)
41: $U(\vec{u}) \leftarrow$ CreateUnknownRelation($\vec{u}$)
42: $\Pi_{inv} \leftarrow \bigwedge_{\mathrm{P}(\vec{x}) \in \Sigma_2}$ Invariant(P($\vec{x}$))
43: $r, \_, \mathcal{A} \leftarrow$ Prove($\varnothing$, $\mathcal{L}$, $\Sigma_1 \wedge U(\vec{u}) \vdash \exists\vec{v}.(\Sigma_2 \wedge \Pi_{inv})$, NoSyn)
44: **if** $r =$ Valid$\langle_\rangle$ **then**
45: $\quad \overline{U}(\vec{u}) \leftarrow$ Solve($\mathcal{A}$, $U(\vec{u})$)
46: $\quad$ **if** $\Sigma_1 \wedge \overline{U}(\vec{u}) \not\equiv false$ **then return** $\{\Sigma_1 \wedge \overline{U}(\vec{u}) \vdash \exists\vec{v}.\Sigma_2\}$
47: **return** $\varnothing$ *//No refined templates are found.*

Fig. 15. Preprocess a lemma template's antecedent

For example, given the template T_3, Preprocess sets up an unknown relation $U(x_1, x_2, x_3, x_4, n_1)$ in the template's antecedent, and introduces the invariant $n_2 \geq 1$ of $\mathsf{dll}(x_5, x_6, x_7, x_8, n_2)$ in the template's consequent to create the following unknown entailment E_{u_2}.

$$E_{u_2} \triangleq \mathsf{dll}(x_1, x_2, x_3, x_4, n_1) \wedge U(x_1, x_2, x_3, x_4, n_1) \vdash \exists x_{5,\ldots,11}, n_2.(\mathsf{dll}(x_5, x_6, x_7, x_8, n_2) * x_9 \mapsto x_{10}, x_{11} \wedge n_2 \geq 1)$$

$$\dfrac{\dfrac{\dfrac{\mathcal{A}_1 \triangleq \{x_1{=}x_3 \wedge n_1{=}1 \wedge U(x_{1,2,3,4}, n_1) \rightarrow false\}}{x_1{=}x_3 \wedge n_1{=}1 \wedge U(x_{1,2,3,4}, n_1) \vdash \exists x_{5,\ldots,11}, n_2.(\mathsf{dll}(x_5, x_6, x_7, x_8, n_2) \wedge x_1{=}x_9 \wedge x_2{=}x_{10} \wedge x_4{=}x_{11} \wedge n_2 \geq 1)}\,\mathsf{U}^2_{\Sigma}}{x_1 \mapsto x_2, x_4 \wedge x_1{=}x_3 \wedge n_1{=}1 \wedge U(x_{1,2,3,4}, n_1) \vdash \exists x_{5,\ldots,11}, n_2.(\mathsf{dll}(x_5, x_6, x_7, x_8, n_2) * x_9 \mapsto x_{10}, x_{11} \wedge n_2 \geq 1)}\,{*}\mapsto \quad \dfrac{\dfrac{\mathcal{A}_2 \triangleq \{U(x_{1,2,3,4}, n_1) \rightarrow \exists x_{5,\ldots,11}, n_2.(x_1{=}x_9 \wedge x_2{=}x_{10} \wedge u{=}x_{11} \wedge u{=}x_5 \wedge x_1{=}x_6 \wedge x_3{=}x_7 \wedge x_4{=}x_8 \wedge n_1{-}1{=}n_2 \wedge n_2 \geq 1)\}}{U(x_{1,2,3,4}, n_1) \vdash \exists x_{5,\ldots,11}, n_2.(x_1{=}x_9 \wedge x_2{=}x_{10} \wedge u{=}x_{11} \wedge u{=}x_5 \wedge x_1{=}x_6 \wedge x_3{=}x_7 \wedge x_4{=}x_8 \wedge n_1{-}1{=}n_2 \wedge n_2 \geq 1)}\,\mathsf{U}^1_{\Pi}}{\mathsf{dll}(u, x_1, x_3, x_4, n_1{-}1) \wedge U(x_{1,2,3,4}, n_1) \vdash \exists x_{5,\ldots,11}, n_2.(\mathsf{dll}(x_5, x_6, x_7, x_8, n_2) \wedge x_1{=}x_9 \wedge x_2{=}x_{10} \wedge u{=}x_{11} \wedge n_2 \geq 1)}\,{*}\mathsf{P}}{x_1 \mapsto x_2, u * \mathsf{dll}(u, x_1, x_3, x_4, n_1{-}1) \wedge U(x_{1,2,3,4}, n_1) \vdash \exists x_{5,\ldots,11}, n_2.(\mathsf{dll}(x_5, x_6, x_7, x_8, n_2) * x_9 \mapsto x_{10}, x_{11} \wedge n_2 \geq 1)}\,{*}\mapsto}{E_{u_2} \triangleq \mathsf{dll}(x_1, x_2, x_3, x_4, n_1) \wedge U(x_1, x_2, x_3, x_4, n_1) \vdash \exists x_{5,\ldots,11}, n_2.(\mathsf{dll}(x_5, x_6, x_7, x_8, n_2) * x_9 \mapsto x_{10}, x_{11} \wedge n_2 \geq 1)}\,\mathsf{ID}$$

Fig. 16. A possible proof tree of the unknown entailment E_{u_2}

This entailment will be proved by induction to collect constraints about the unknown relation U. We present its detailed proof in Figure 16. Observe that the assumption constraint $U(x_1, x_2, x_3, x_4, n_1) \rightarrow \exists x_5, \ldots, x_{11}, n_2.(x_1{=}x_9 \wedge x_2{=}x_{10} \wedge u{=}x_{11} \wedge u{=}x_5 \wedge x_1{=}x_6 \wedge x_3{=}x_7 \wedge x_4{=}x_8 \wedge n_1{-}1{=}n_2 \wedge n_2 \geq 1)$ in $\mathcal{A}_2$ can be simplified by eliminating existentially quantified variables to become $U(x_1, x_2, x_3, x_4, n_1) \rightarrow n_1 \geq 2$. Therefore, we can obtain a set $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ containing the following unknown assumptions:

(1) $x_1{=}x_3 \wedge n_1{=}1 \wedge U(x_1, x_2, x_3, x_4, n_1) \rightarrow false$ (2) $U(x_1, x_2, x_3, x_4, n_1) \rightarrow n_1 \geq 2$

The procedure Preprocess can solve the *full assumption constraint set* $\mathcal{A}$ to discover a potential solution $\overline{U}(x_1, x_2, x_3, x_4, n_1) \equiv n_1 \geq 2$, which allows T_3 to be refined to a new lemma template T_3':

$$T_3' \triangleq \mathsf{dll}(x_1, x_2, x_3, x_4, n_1) \wedge n_1 \geq 2 \vdash \exists x_5, \ldots, x_{11}, n_2.(\mathsf{dll}(x_5, x_6, x_7, x_8, n_2) * x_9 \mapsto x_{10}, x_{11})$$

5.4.2 Fine-Tuning Lemma Templates' Consequents. This step aims to refine further the consequent of the lemma template discovered in the preprocessing phase. The refinement is performed by the recursive procedure FineTuneConseq (Figure 17). Its input is a (refined) lemma template $\Sigma_1 \wedge \Pi_1 \vdash \exists\vec{v}.(\Sigma_2 \wedge \Pi_2)$. When FineTuneConseq is invoked in the first time by RefineConseq, Π_2 is set to *true* (Figure 14, line 38).

Initially, FineTuneConseq establishes an unknown relation $U(\vec{u})$ on all variables in the templates and creates an unknown entailment $\Sigma_1 \wedge \Pi_1 \vdash \exists\vec{v}.(\Sigma_2 \wedge \Pi_2 \wedge U(\vec{u}))$ (lines 48, 49). Then, it proves the

unknown entailment by induction to collect a set $\mathcal{A}$ of unknown assumptions (line 50). Similarly to the antecedent refinement (Section 5.3), $\mathcal{A}$ will be *exhaustively solved*: the entire set and its subsets will be examined until a feasible solution is found to obtain a valid inductive lemma (lines 52 – 59).

We also prevent the proof system from entering nested lemma synthesis phases, controlled by the argument NoSyn, when Prove is invoked to collect the unknown assumption set $\mathcal{A}$ (line 50) or to verify the inductive lemma (line 55). Witness proof tree of a candidate lemma, if having, is examined for the occurrence of the induction hypothesis application rule (IH), to determine if this candidate is a valid inductive lemma (line 56).

However, unlike the antecedent refinement, FineTuneConseq keeps refining the template until its consequent cannot be refined further (lines 56, 59). This repetition ensures that the returned lemma's consequent, if discovered, is as strong as possible.

Procedure FineTuneConseq($\mathcal{L}$, $\Sigma_1 \wedge \Pi_1 \vdash \exists \vec{v}.(\Sigma_2 \wedge \Pi_2)$)

Input: $\Sigma_1 \wedge \Pi_1 \vdash \exists \vec{v}.(\Sigma_2 \wedge \Pi_2)$ is a lemma template, $\mathcal{L}$ is a valid lemma set
Output: a set of at most one synthesized lemma
Requires: $valid(\mathcal{L})$
Ensures: $(res = \varnothing) \vee \exists\, \Pi_2'.(res = \{\Sigma_1 \wedge \Pi_1 \vdash \exists \vec{v}.(\Sigma_2 \wedge \Pi_2 \wedge \Pi_2')\} \wedge valid(res))$

48: $\vec{u} \leftarrow$ FreeVars$(\Sigma_1, \Sigma_2, \Pi_1, \Pi_2)$
49: $U(\vec{u}) \leftarrow$ CreateUnknownRelation$(\vec{u})$
50: $r, \_, \mathcal{A} \leftarrow$ Prove$(\varnothing, \mathcal{L}, \Sigma_1 \wedge \Pi_1 \vdash \exists \vec{v}.(\Sigma_2 \wedge \Pi_2 \wedge U(\vec{u})),$ NoSyn$)$
51: **if** $r =$ Valid$\langle_\rangle$ **then**
52: **for each** $\mathcal{A}'$ **in** SuperSet$(\mathcal{A})$ **do**
53: $\overline{U}(\vec{u}) \leftarrow$ Solve$(\mathcal{A}', U(\vec{u}))$
54: $L \leftarrow (\Sigma_1 \wedge \Pi_1 \vdash \exists \vec{v}.(\Sigma_2 \wedge \Pi_2 \wedge \overline{U}(\vec{u}))$
55: $r', \_, \_ \leftarrow$ Prove$(\varnothing, \mathcal{L}, L,$ NoSyn$)$ *// Verify ...*
56: **if** $r' =$ Valid$\langle\xi\rangle$ **and** IH $\in \xi$ **then** *// the inductive lemma, and ...*
57: $\mathcal{L}_{\text{syn}} \leftarrow$ FineTuneConseq$(\mathcal{L}, L)$ *// strengthen its consequent.*
58: **if** $\mathcal{L}_{\text{syn}} = \varnothing$ **then return** $\{L\}$
59: **else return** $\mathcal{L}_{\text{syn}}$
60: **return** $\varnothing$ *// No lemmas are found.*

Fig. 17. Fine-tuning the consequent of a lemma template

For example, given the preprocessed template T_3', FineTuneConseq creates an unknown relation $U(x_1, \ldots, x_{11}, n_1, n_2)$, or $U(\vec{x}, \vec{n})$ for short, involving all variables in T_3' and constructs the unknown entailment E_{u_3}. This entailment will be proved again to collect constraints about $U(\vec{x}, \vec{n})$.

$$E_{u_3} \triangleq \mathsf{dll}(x_1, x_2, x_3, x_4, n_1) \wedge n_1 \geq 2 \vdash \exists x_5, \ldots, x_{11}, n_2.(\mathsf{dll}(x_5, x_6, x_7, x_8, n_2) * x_9 \mapsto x_{10}, x_{11} \wedge U(\vec{x}, \vec{n}))$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{A}_1}\ \ldots}}{\mathsf{dll}(u, x_1, x_3, x_4, n_1{-}1) \wedge n_1 \geq 2 \vdash \exists x_{5,\ldots,11}, n_2, v.(\mathsf{dll}(v, x_5, x_7, x_8, n_2{-}1) * x_9 \mapsto x_{10}, x_{11} \wedge U(\vec{x}, \vec{n}) \wedge x_1{=}x_5 \wedge x_2{=}x_6 \wedge u{=}v \wedge v{=}x_7 \wedge n_2{-}1{=}1)}\ \mathsf{P_R}}{x_1 \mapsto x_2, u * \mathsf{dll}(u, x_1, x_3, x_4, n_1{-}1) \wedge n_1 \geq 2 \vdash \exists x_{5,\ldots,11}, n_2, v.(x_5 \mapsto x_6, v * \mathsf{dll}(v, x_5, x_7, x_8, n_2{-}1) * x_9 \mapsto x_{10}, x_{11} \wedge U(\vec{x}, \vec{n}))}\ *\mapsto}{x_1 \mapsto x_2, u * \mathsf{dll}(u, x_1, x_3, x_4, n_1{-}1) \wedge n_1 \geq 2 \vdash \exists x_{5,\ldots,11}, n_2.(\mathsf{dll}(x_5, x_6, x_7, x_8, n_2) * x_9 \mapsto x_{10}, x_{11} \wedge U(\vec{x}, \vec{n}))}\ \mathsf{P_R}}{\mathsf{dll}(u, x_1, x_3, x_4, n_1{-}1) \wedge n_1 \geq 2 \vdash \exists x_{5,\ldots,11}, n_2, v.(v \mapsto x_5, x_8 * x_9 \mapsto x_{10}, x_{11} \wedge U(\vec{x}, \vec{n}) \wedge x_1{=}x_5 \wedge x_2{=}x_6 \wedge u{=}v \wedge v{=}x_7 \wedge n_2{-}1{=}1)}\ \mathsf{ID}}{\mathsf{dll}(r, u, x_3, x_4, n_1{-}2) \wedge n_1 \geq 2 \vdash \exists x_{5,\ldots,11}, n_2, v.(x_9 \mapsto x_{10}, x_{11} \wedge U(\vec{x}, \vec{n}) \wedge x_1{=}x_5 \wedge x_2{=}x_6 \wedge u{=}v \wedge v{=}x_7 \wedge r{=}x_8 \wedge n_2{-}1{=}1)}\ *\mapsto}{r \mapsto u, x_4 \wedge n_1 \geq 2 \wedge r{=}x_3 \wedge n_1{-}2{=}1 \vdash \exists x_{5,\ldots,11}, n_2, v.(x_9 \mapsto x_{10}, x_{11} \wedge U(\vec{x}, \vec{n}) \wedge x_1{=}x_5 \wedge x_2{=}x_6 \wedge u{=}v \wedge v{=}x_7 \wedge r{=}x_8 \wedge r{=}x_9 \wedge u{=}x_{10} \wedge x_4{=}x_{11} \wedge n_2{-}1{=}1)}\ *\mapsto}{n_1 \geq 2 \wedge r{=}x_3 \wedge n_1{-}2{=}1 \vdash \exists x_{5,\ldots,11}, n_2, v.(U(\vec{x}, \vec{n}) \wedge x_1{=}x_5 \wedge x_2{=}x_6 \wedge u{=}v \wedge v{=}x_7 \wedge r{=}x_8 \wedge r{=}x_9 \wedge u{=}x_{10} \wedge x_4{=}x_{11} \wedge n_2{-}1{=}1)}\ \mathsf{U}^1_\Pi}{\mathcal{A}_3 \triangleq \{n_1{\geq}2 \wedge r{=}x_3 \wedge n_1{-}2{=}1 \rightarrow \exists x_{5,\ldots,11}, n_2, v.(U(\vec{x}, \vec{n}) \wedge x_1{=}x_5 \wedge x_2{=}x_6 \wedge u{=}v \wedge v{=}x_7 \wedge r{=}x_8 \wedge r{=}x_9 \wedge u{=}x_{10} \wedge x_4{=}x_{11} \wedge n_2{-}1{=}1)\}}$$

$\mathcal{A}_2$ — … $\quad$ $\mathcal{A}_4$ — …

Fig. 18. A partial proof tree of the unknown entailment E_{u_3}

We present a partial proof tree of E_{u_3} in Figure 18, where FineTuneConseq is able to collect a set of unknown assumptions $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3 \cup \mathcal{A}_4$. More specifically, $\mathcal{A}_3 \triangleq \{n_1{\geq}2 \wedge r{=}x_3 \wedge n_1{-}2{=}1 \vdash \exists x_{5,\ldots,11}, n_2, v.\ (U(\vec{x},\vec{n}) \wedge x_1{=}x_5 \wedge x_2{=}x_6 \wedge u{=}v \wedge v{=}x_7 \wedge r{=}x_8 \wedge r{=}x_9 \wedge u{=}x_{10} \wedge x_4{=}x_{11} \wedge n_2{-}1{=}1)\}$ is an assumption subset derived from a potential proof path of an inductive lemma, and $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_4$ relate to other proof paths. The set $\mathcal{A}$ can be partially solved by considering only $\mathcal{A}_3$. Here, a possible solution of $\mathcal{A}_3$ is $\overline{U}_p(x_1, ..., x_{11}, n_1, n_2) \equiv x_1{=}x_5 \wedge x_2{=}x_6 \wedge x_3{=}x_8 \wedge x_4{=}x_{11} \wedge x_7{=}x_{10} \wedge x_8{=}x_9$. This solution can be substituted back to E_{u_3} to derive the refined lemma template T_3''.

$$T_3'' \triangleq \mathsf{dll}(x_1, x_2, x_3, x_4, n_1) \wedge n_1{\geq}2 \vdash \exists x_7, n_2.(\mathsf{dll}(x_1, x_2, x_7, x_3, n_2) * x_3{\mapsto}x_7, x_4)$$

Then, FineTuneConseq constructs another unknown entailment $U'(x_1, x_2, x_3, x_4, x_7, n_1, n_2)$ on all variables of T_3'' and creates a new unknown entailment, which will be proved again to find a solution $\overline{U}_p'(x_1, x_2, x_3, x_4, x_7, n_1, n_2) \equiv n_1{=}n_2{+}1$. This solution helps to refine the template T_3'' to obtain an inductive lemma $\overline{T}_3 \triangleq \mathsf{dll}(x_1, x_2, x_3, x_4, n_1) \wedge n_1{\geq}2 \vdash \exists x_7.(\mathsf{dll}(x_1, x_2, x_7, x_3, n_1{-}1) * x_3{\mapsto}x_7, x_4)$, which can be equivalently transformed to the motivating lemma L_3 in Section 2.

5.5 Inferring Pure Invariants of Inductive Heap Predicates

We present in this subsection the construction of inductive heap predicates' invariants, which are mainly utilized to preprocess the lemma templates' antecedents (Section 5.4). Our construction is inspired from separation logic literature [Brotherston et al. 2014; Chin et al. 2012; Le et al. 2016].

Definition 5.2 (Invariant of inductive heap predicates). Given an inductive heap predicate $\mathsf{P}(\vec{x})$, a pure formula $\mathsf{Invariant}(\mathsf{P}(\vec{x}))$ is called an invariant of $\mathsf{P}(\vec{x})$ iff $s, h \vDash \mathsf{P}(\vec{x})$ implies that $s \vDash \mathsf{Invariant}(\mathsf{P}(\vec{x}))$, for all model s, h. Formally, $\forall s, h.\ (s, h \vDash \mathsf{P}(\vec{x}) \rightarrow s \vDash \mathsf{Invariant}(\mathsf{P}(\vec{x})))$.

Constructing Pure Invariants. We also exploit a template-based approach to discover the pure invariants. For a system of k (mutually) inductive heap predicates $\mathsf{P}_1(\vec{x}_1), \ldots, \mathsf{P}_k(\vec{x}_k)$, we first create k unknown relations $U_{\mathsf{P}_1}(\vec{x}_1), \ldots, U_{\mathsf{P}_k}(\vec{x}_k)$ to respectively represent their desired invariants. Then, we follow the predicates' definitions to establish constraints about the unknown relations.

In particular, consider each definition case $F_j^i(\vec{x}_i)$ of a predicate $\mathsf{P}_i(\vec{x}_i)$, which is of the form $F_j^i(\vec{x}_i) \triangleq \exists \vec{u}.(\Sigma_j^i \wedge \Pi_j^i)$. Suppose that $\mathsf{Q}_1(\vec{u}_1), ..., \mathsf{Q}_n(\vec{u}_n)$ are *all* inductive heap predicates in Σ_j^i, where $\mathsf{Q}_1, ..., \mathsf{Q}_n \in \{\mathsf{P}_1, ..., \mathsf{P}_k\}$, we create the following unknown assumption:

$$U_{\mathsf{Q}_1}(\vec{u}_1) \wedge \ldots \wedge U_{\mathsf{Q}_n}(\vec{u}_n) \wedge \Pi_j^i \quad \rightarrow \quad U_{\mathsf{P}_i}(\vec{x}_i)$$

Thereafter, the system of all unknown assumptions can be solved by the constraint solving technique based on Farkas' lemma (Section 5.6) to discover the actual invariants of $\mathsf{P}_1(\vec{x}_1), \ldots, \mathsf{P}_k(\vec{x}_k)$.

For example, given the predicate $\mathsf{dll}(hd, pr, tl, nt, len)$ in Section 2, whose definition is

$$\mathsf{dll}(hd, pr, tl, nt, len) \stackrel{\text{def}}{=} (hd{\mapsto}pr, nt \wedge hd{=}tl \wedge len{=}1) \vee \exists u.(hd{\mapsto}pr, u * \mathsf{dll}(u, hd, tl, nt, len{-}1))$$

We create an unknown relation $U_{\mathsf{dll}}(hd, pr, tl, nt, len)$ representing its invariant and follow the definition of $\mathsf{dll}(hd, pr, tl, nt, len)$ to establish the following constraints:

$$(1)\ hd{=}tl \wedge len{=}1 \rightarrow U_{\mathsf{dll}}(hd, pr, tl, nt, len) \qquad (2)\ U_{\mathsf{dll}}(u, hd, tl, nt, len{-}1) \rightarrow U_{\mathsf{dll}}(hd, pr, tl, nt, len)$$

We then solve these constraints to obtain the solution $\overline{U}_{\mathsf{dll}}(hd, pr, tl, nt, len) \equiv len \geq 1$, which is also the invariant of $\mathsf{dll}(hd, pr, tl, nt, len)$.

5.6 Solving Assumption Constraints with Farkas' Lemma

In this subsection, we describe the underlying constraint solving technique based on Farkas' lemma. This technique is implemented in the procedure Solve, which is frequently invoked in the lemma synthesis to solve an unknown assumption set (Sections 5.3, 5.4, and 5.5). We will formally restate Farkas' lemma and explain how it is applied in constraint solving.

Farkas' Lemma [Schrijver 1986]. Given a conjunction of linear constraints $\bigwedge_{j=1}^{m} \sum_{i=1}^{n} a_{ij}x_i + b_j \geq 0$, which is satisfiable, and a linear constraint $\sum_{i=1}^{n} c_i x_i + \gamma \geq 0$, we have:

$$\forall x_1 \ldots x_n. \left(\bigwedge_{j=1}^{m} \sum_{i=1}^{n} a_{ij}x_i + b_j \geq 0 \right) \rightarrow \sum_{i=1}^{n} c_i x_i + \gamma \geq 0 \quad \text{iff} \quad \exists \lambda_1 \ldots \lambda_m \geq 0. \left(\bigwedge_{i=1}^{n} c_i = \sum_{j=1}^{m} \lambda_j a_{ij} \wedge \sum_{j=1}^{m} \lambda_j b_j \leq \gamma \right)$$

Solving Constraints. Given an assumption set $\mathcal{A}$ of an unknown relation $U(x_1, \ldots, x_m, u_1, \ldots, u_n)$, where $x_1, \ldots, x_m$ are spatial variables and $u_1, \ldots, u_n$ are integer variables, the procedure Solve aims to find an actual definition $\overline{U}$ of U in the following template, where c_{ij}, d_{ij} are unknown coefficients, M, N are pre-specified numbers of conjuncts.

$$U(x_1, \ldots, x_m, u_1, \ldots, u_n) \quad \triangleq \quad \left(\bigwedge_{j=1}^{M} \sum_{i=1}^{m} c_{ij}x_i + c_{0j} \geq 0 \right) \wedge \left(\bigwedge_{j=1}^{N} \sum_{i=1}^{n} d_{ij}u_i + d_{0j} \geq 0 \right)$$

Recall that our lemma synthesis framework can incrementally discover the final solution $\overline{U}$ in several passes. Hence, we can set M, N small, e.g., $1 \leq M, N \leq 3$, or $M=0$, $N \leq 6$ (no spatial contraints), or $M \leq 6$, $N=0$ (no arithmetic constraints) to make the constraint solving more effective. We restrict the coefficients c_{ij} to represent equality or disequality constraints of spatial variables, e.g., $x_k=x_l$, $x_k \neq x_l$, $x_k=0$, and $x_k \neq 0$, where 0 denotes nil. The constraint $x_k=x_l$ can be encoded as $x_k - x_l \geq 0 \wedge -x_k + x_l \geq 0$, or the encoding of $x_k \neq x_l$ is $x_k - x_l - 1 \geq 0$. Therefore, it requires that $-1 \leq c_{ij} \leq 1$ for $i>0$, and $c_{0j}=0$ (for equalities) or $c_{0j}=-1$ (for disequalities). We also add the restrictions $-1 \leq \sum_{i=1}^{m} c_{ij} \leq 1$ and $1 \leq \sum_{i=1}^{m} |c_{ij}| \leq 2$ to ensure that the spatial constraints involve at most two variables.

In summary, the assumption set $\mathcal{A}$ of the unknown relation U can be solved in three steps:

- Normalize assumptions in the set $\mathcal{A}$ into the form of Horn clause, and substitute all occurrences of U in the normalized assumptions by its template to obtain a set of normalized constraints.
- Apply Farkas' lemma to eliminate universal quantification to obtain new constraints with only existential quantification over the unknown coefficients c_{ij}, d_{ij} and the factors λ_j.
- Solve the new constraints by an off-the-shelf prover, such as Z3 [Moura and Bjørner 2008], to find the concrete values of the unknown coefficients, thus discover the actual definition of U.

5.7 Soundness of the Lemma Synthesis Framework

We claim that our lemma synthesis framework is sound, that is, all lemmas synthesized by the framework are semantically valid. We formally state the soundness in the following Theorem 5.3.

Theorem 5.3 (Soundness of the lemma synthesis). *Given a normal goal entailment E which does not contain any unknown relation and a set of valid input lemma $\mathcal{L}$, if the lemma synthesis procedure* SynthesizeLemma *returns a set of lemmas $\mathcal{L}_{\text{syn}}$, then all lemmas in $\mathcal{L}_{\text{syn}}$ are semantically valid.*

Proof. Figure 11 shows that all lemmas returned by SynthesizeLemma are directly discovered by RefineAnte and RefineConseq. In addition, all lemmas returned by RefineConseq are realized by FineTuneConseq (Figure 14, line 38). Moreover, all lemmas discovered by RefineAnte and FineTuneConseq are verified by Prove in a setting that disables the lemma synthesis (NoSyn) and utilizes the valid lemma set $\mathcal{L}$ (Figure 12, line 31 and Figure 17, line 55). It follows from Proposition 4.2 that if Prove returns Valid⟨_⟩ when verifying a lemma, then the lemma is semantically valid. Consequently, all lemmas returned by SynthesizeLemma are semantically valid. □

6 EXPERIMENTS

We have implemented the lemma synthesis framework into a prototype prover, named **SLS**[3] and have conducted two experiments to evaluate its ability in proving entailments. Both the prover and the experiment details are available online at https://songbird-prover.github.io/lemma-synthesis.

[3] **SLS** is built on top of an existing prover **Songbird** [Ta et al. 2016]; its name stands for "**S**ongbird + **L**emma **S**ynthesis"

Table 1. Evaluation on the existing entailment benchmarks, where participants are **Slide** (Sld), **Sleek** (Slk), **Spen** (Spn), **Cyclist** (Ccl), **Songbird** (Sbd) and our prototype prover **SLS**

Benchmark		Proved Entailments							Total Proving Time (s)						Average Time (s)						Lemma Syn				Lemma App		
Category		#En	Sld	Slk	Spn	Ccl	Sbd	Sls	Sld	Slk	Spn	Ccl	Sbd	Sls	Sld	Slk	Spn	Ccl	Sbd	Sls	#Lm	T (s)	A (s)	O (%)	#Cv	#Sp	#Cb
sll_entl	bolognesa	57	0	0	**57**	0	**57**	**57**	0.0	0.0	23.6	0.0	140.3	18.5	–	–	0.41	–	2.46	0.32	0	0.0	–	0.0	0	0	0
	clones	60	0	**60**	**60**	**60**	**60**	**60**	0.0	3.7	3.5	0.5	3.9	0.7	–	0.06	0.06	0.01	0.07	0.01	0	0.0	–	0.0	0	0	0
	smallfoot	54	0	**54**	**54**	**54**	**54**	**54**	0.0	2.7	2.5	11.8	3.5	4.2	–	0.05	0.05	0.22	0.06	0.08	0	0.0	–	0.0	0	0	0
slrd_entl	singly-ll	64	12	48	3	63	**64**	**64**	1.0	6.3	0.1	2.1	8.3	1.7	0.08	0.13	0.04	0.03	0.13	0.03	0	0.0	–	0.0	0	0	0
	doubly-ll	37	14	17	9	29	25	**35**	38.3	3.1	0.4	112.5	11.3	91.9	2.74	0.18	0.04	3.88	0.45	2.63	18[10]	51.2	2.8	55.7	8 (68)	0	2 (2)
	nested-ll	11	0	5	**11**	7	**11**	**11**	0.0	2.2	0.5	16.7	2.3	0.4	–	0.44	0.04	2.38	0.21	0.04	0	0.0	–	0.0	0	0	0
	skip-list	13	0	4	**13**	5	**13**	**13**	0.0	1.1	1.1	0.6	8.1	1.3	–	0.27	0.08	0.11	0.63	0.10	0	0.0	–	0.0	0	0	0
	tree	26	12	14	0	23	23	**24**	110.0	2.9	0.0	58.8	11.5	2.2	9.16	0.21	–	2.55	0.50	0.09	0	0.0	–	0.0	0	0	0
slrd_indt	ll/ll2	24	0	0	0	**24**	**24**	**24**	0.0	0.0	0.0	60.2	11.9	78.5	–	–	–	2.51	0.50	3.27	14[10]	12.6	0.9	16.1	4 (4)	0	6 (11)
	ll-even/odd	20	0	0	0	**20**	**20**	**20**	0.0	0.0	0.0	34.6	50.3	1.3	–	–	–	1.73	2.52	0.07	0	0.0	–	0.0	0	0	0
	ll-left/right	20	0	0	0	**20**	**20**	**20**	0.0	0.0	0.0	19.5	10.6	1.3	–	–	–	0.97	0.53	0.06	0	0.0	–	0.0	0	0	0
	misc.	32	0	0	0	31	**32**	**32**	0.0	0.0	0.0	254.2	55.0	19.5	–	–	–	8.20	1.72	0.61	2[2]	1.3	0.7	6.7	2 (3)	0	0
Total		418	38	202	207	336	403	**414**	149.2	21.9	31.6	571.5	317.2	221.6	3.93	0.11	0.15	1.70	0.79	0.54	34[22]	65.2	1.9	29.4	14 (75)	0	8 (13)

The first experiment was done on literature benchmarks, which include sll_entl, slrd_entl from the separation logic competition SL-COMP 2014 [Sighireanu and Cok 2016], and slrd_indt by Ta et al. [2016]. These benchmarks focus on representing the data structures' shapes, hence, their pure formulas solely contain equality constraints among spatial variables. Therefore, we decided to compose a richer entailment benchmark slrd_lm, which captures also arithmetic constraints of the data structures' size and content, and then performed the second experiment. Moreover, we only considered entailments that are marked as *valid*. Our experiments were conducted on a Ubuntu 14.04 LTS machine with CPU Intel® Core™ i7-6700 (3.4GHz) and RAM 16GB. We compared **SLS** against state-of-the-art separation logic provers, including **Slide** [Iosif et al. 2013], **Sleek** [Chin et al. 2012], **Spen** [Enea et al. 2014], **Cyclist** [Brotherston et al. 2011] and **Songbird** [Ta et al. 2016]. These provers were configured to run with a timeout of 180 seconds for each entailment.

We present the first experiment in Table 1. The benchmark sll_entl relates to only singly linked list and is categorized by its original sub-benchmarks. The two benchmarks slrd_entl and slrd_indt are classified based on the related data structures, which are variants of linked lists and trees. We report in each category the number of *entailments successfully proved* by each prover, where the best results are in **bold** (the total number of entailments is shown in the column **#En**). We also report the *total* and the *average time* in seconds of each prover for all *proved entailments*. To be fair, time spent on *unproved entailments* are *not considered* since a prover might spend up to the timeout of 180(s) for each such entailment. We also provide the statistics of how lemmas are synthesized and applied by **SLS**. The numbers of lemmas synthesized and used are presented in the column **#Lm**, where $x[y]$ means that there are totally x lemmas synthesized, and only y of them are successfully used. The *total* and the *average time* spent on synthesizing all lemmas are displayed in the columns **T** and **A**. The synthesis *overhead* is shown in the column **O**, which is the percentage (%) of the total synthesizing time over the total proving time. We also classify the synthesized lemmas into three groups of *conversion*, *split*, and *combination* lemmas (**#Cv**, **#Sp**, and **#Cb**), in a spirit similar to the motivating lemmas L_1, L_2 and L_3 (Section 2). The number $x(y)$ in each group indicates that there are x lemmas applied, and they are repeatedly applied y times.

Table 1 shows that **SLS** can prove *most* of the entailments in existing benchmarks (414/418 ≈ 99%) on an average of 0.54 seconds per entailment, which is 1.46 times faster than the second best prover **Songbird**. Moreover, **SLS** outperforms the third best prover **Cyclist** in both the number of proved entailments (more than 78 entailments) and the average proving time (3.15 times faster). Other provers like **Sleek**, **Spen** and **Slide** can prove no more than 50% of the entailments that **SLS** can prove. In addition, **SLS** achieves the best results in *all* categories, which demonstrates

Table 2. Evaluation on the benchmark slrd_lm, where • marks categories with arithmetic constraints

Benchmark slrd_lm	Proved Entails				Proving Time (s)			Average Time (s)			Lemma Synthesis				Lemma Application		
Category	#En	Ccl	Sbd	Sls	Ccl	Sbd	Sls	Ccl	Sbd	Sls	#Lm	T (s)	A (s)	O (%)	#Cv	#Sp	#Cb
ll/rev	20	17	18	**20**	446.5	87.1	205.0	26.27	4.84	10.25	28 [20]	8.1	0.3	3.9	18 (25)	0	2 (2)
ll-even/odd	17	**17**	**17**	**17**	3.5	0.6	0.4	0.21	0.03	0.02	0	0.0	–	0.0	0	0	0
ll/rev+arith•	72	0	10	**72**	0.0	0.4	203.2	–	0.04	2.82	81 [81]	50.9	0.6	25.1	0	81 (143)	0
ll-sorted•	25	0	3	**19**	0.0	0.1	8.7	–	0.03	0.46	14 [14]	4.9	0.3	55.8	0	0	14 (30)
dll/rev/null	22	**22**	**22**	**22**	8.4	17.6	149.8	0.38	0.80	6.81	12 [7]	59.1	4.9	39.5	7 (8)	0	0
dll/rev/null+arith•	94	0	17	**94**	0.0	27.6	2480.6	–	1.63	26.39	236 [158]	1902.7	8.1	76.7	74 (74)	69 (103)	15 (15)
ll/dll-mixed	5	**5**	**5**	**5**	0.4	0.1	0.2	0.07	0.03	0.03	0	0.0	–	0.0	0	0	0
ll/dll-mixed+arith•	15	0	5	**15**	0.0	0.4	328.3	–	0.08	21.89	12 [10]	227.2	18.9	69.2	8 (8)	2 (2)	0
tree/tseg	18	7	7	**18**	1.3	0.3	135.0	0.18	0.04	7.50	18 [12]	3.8	0.2	2.8	8 (11)	0	4 (5)
tree/tseg+arith•	12	0	3	**12**	0.0	3.5	513.7	–	1.15	42.81	20 [16]	211.2	10.6	41.1	8 (8)	4 (5)	4 (4)
Total	300	68	107	**294**	460.0	137.7	4024.8	6.76	1.29	13.69	421 [318]	2467.7	5.9	61.3	123 (134)	156 (253)	39 (56)

the effectiveness of our lemma synthesis technique. Regarding the 4 entailments that **SLS** cannot prove, they are related to the doubly linked list or the tree data structures (doubly-ll/tree), where the needed lemmas are too complicated to be synthesized within a timeout of 180 seconds.

In the first experiment, the lemma synthesis is only triggered in some categories with an overhead of 29.4%. Although this overhead is considerable, it does not overwhelm the overall proving process since much of the proving time is saved via the lemma application. The lemma synthesis's *efficacy*, determined by the ratio of the number of lemmas applied to the number of lemmas synthesized, is about 65% (22 lemmas used in the total of 34 lemmas synthesized). More interestingly, these 22 lemmas were applied totally 88 times. This fact implies that a lemma can be (re)used multiple times. In this experiment, the conversion lemmas is applied more often than other lemmas (75/88 times).

In the second experiment, we apply the best three provers in the first experiment, i.e., **Cyclist**, **Songbird**, and **SLS**, on the more challenging benchmark slrd_lm. This benchmark was constructed by enhancing the existing inductive heap predicates with richer *numeric* properties of the data structures' size and content. These new predicates enable more challenging entailments to be designed, such as the motivating entailment E_1 from Section 2.

$$E_1 \triangleq \mathsf{dllrev}(x, y, u, v, n) * \mathsf{dll}(v, u, z, t, 200) \wedge n{\geq}100 \vdash \exists r.(\mathsf{dll}(x, y, r, z, n{+}199) * z{\mapsto}r, t)$$

Details of the second experiment is presented in Table 2. Entailments in the benchmark slrd_lm are categorized based on the participating heap predicates and their numeric properties. To minimize the table's size, we group various heap predicates related to the same data structure in one category. For example, the category dll/rev/null contains entailments about doubly linked lists, including normal lists (dll), reversed lists (dllrev), or null-terminated lists (dllnull). Entailments in the category ll/dll-mixed involve both the singly and the doubly linked lists.

Table 2 shows that **SLS** outperforms other provers at *all* categories. In total, **SLS** can prove 98% of the benchmark problems (294/300 entailments), which is 2.7 times better than the second best prover **Songbird** (107/300 entailments). More impressively, in the 5 categories with arithmetic constraints, marked by •, **SLS** can prove more than 5.5 times the number of entailments that **Songbird** can (212 vs. 38 entailments). On the other hand, **Cyclist** performs poorly in this experiment because it does not support numeric properties yet. However, among 82 non-arithmetic entailments, **Cyclist** can prove 68 entailments (about 80%), whereas **SLS** can prove all of them.

In the second experiment, the lemma synthesis overhead is higher (61.3%) since there are many lemmas synthesized (421). However, the overall efficacy is also improved (75.5%) when 318 synthesized lemmas are actually used to prove goal entailments. It is worth noticing that 100% lemmas synthesized in the two arithmetic categories ll/rev+arith and ll-sorted are utilized; this shows the usefulness of our proposed framework in proving sophisticated entailments. In this experiment, the

split lemmas are synthesized (49.1% of the total synthesized lemmas) and prominently used (57% of the total number of lemma application). This interesting fact shows that the slrd_lm benchmark, though handcrafted, was well designed to complement the existing benchmarks.

7 RELATED WORK

There have been various approaches proposed to prove separation logic entailments. A popular direction is to restrict the inductive heap predicates to certain classes such as: predicates whose syntax and semantics are defined beforehand [Berdine et al. 2004, 2005b; Bozga et al. 2010; Pérez and Rybalchenko 2011, 2013; Piskac et al. 2013, 2014], predicates describing *variants of linked lists* [Enea et al. 2014], or predicates satisfying a particular *bounded tree width property* [Iosif et al. 2013, 2014]. These restrictions enable the invention of practical and effective entailment proving techniques. However, these predicate classes *cannot model sophisticated constraints* of data structures, which involve not only the shape but also the size or the content, like the predicates dll and dllrev in Section 2. In addition, the existing techniques are tied to fixed sets of inductive heap predicates and *could not be automatically extended* to handle new predicates. This extension requires extra efforts.

Another research direction is to focus on a broader class of *user-defined inductive heap predicates*. In particular, Chin et al. [2012] proposed a proof system based on the *unfold-and-match* technique: heap predicates in a goal entailment can be *unfolded* by their definitions to produce possibly identical predicates in the antecedent and consequent, which can be *matched* and *removed* to derive simpler sub-goal entailments. However, an inductive heap predicate can be unfolded infinitely often, which leads to the infinite derivation of an entailment proof. To deal with such situation, Brotherston et al. [2011] and Chu et al. [2015] proposed inspiring techniques respectively based on cyclic and induction proofs where *the infinite unfolding sequences can be avoided by induction hypothesis applications*. In their works, the induction hypotheses are discovered directly from the candidate entailments; they might not be sufficiently general to prove sophisticated entailments. Therefore, these entailments' proofs often require the supporting lemmas, such as L_1, L_2, L_3 (Section 2). These lemmas are also needed by other non-induction based verification systems, such as [Chin et al. 2012; Qiu et al. 2013]. At present, these systems require users to manually provide the lemmas.

To the best of our knowledge, there have been two approaches aiming to automatically discover the supporting lemmas. The first approach is the *mutual induction* proof presented in [Ta et al. 2016]. This work speculates lemmas from *all entailments* which are already derived in an ongoing induction proof to assist in proving future entailments introduced within the same proof. This speculation provides more lemma/induction hypothesis candidates than the cyclic-based [Brotherston et al. 2011] and induction-based [Chu et al. 2015] techniques. Consequently, it can increase the chance of successfully proving the entailments. However, the mutual induction proof *cannot handle* sophisticated entailments, such as E_1 in Section 2. All entailments derived from E_1 may contain specific constraints and cannot be applied to prove other derived entailments.

The second approach is the *lemma generation* presented in [Enea et al. 2015]. This work considers an interesting class of inductive heap predicates satisfying the notions of *compositionality* and *completion*. Under these properties, a class of lemmas can be enumerated beforehand, either to *convert* inductive predicates *of the same arity*, or to *combine* two inductive predicates. However, this technique *cannot generate* lemmas which convert predicates of different arities, or combine a singleton heap predicate with an inductive heap predicate, or split an inductive heap predicate into other predicates. In addition, an inductive heap predicate satisfying the *compositionality* property has exactly one base-case definition, whose heap part is also empty. Moreover, each inductive-case definition must contain a singleton heap predicate whose *root* address is one of the inductive heap predicate's

arguments, like the *compositional* predicate $\mathsf{ls}(x, y) \stackrel{\text{def}}{=} (x{=}y) \vee \exists u.(x{\mapsto}u * \mathsf{ls}(u, y))$. This technique, therefore, cannot generate lemmas for predicates with non-empty base cases, e.g., dll and dllrev in Section 2, or lemmas for the predicates defined in a *reverse-fashion*, like the *non-compositional* predicate $\mathsf{lsrev}(x, y) \stackrel{\text{def}}{=} (x{=}y) \vee \exists u.(\mathsf{lsrev}(x, u) * u{\mapsto}y)$. These *reverse-fashion* predicates are prevalent in SL-COMP 2014's benchmarks, such as RList, ListO, DLL_plus, DLL_plus_rev, DLL_plus_mid.

In the *synthesis* context, there is an appealing approach called SyGuS, a.k.a., Syntax-Guided Synthesis [Alur et al. 2015]. This approach aims to *infer computer programs* satisfying certain restrictions on its syntax and semantics (respectively constrained by a context free grammar and a SMT formula). Techniques following SyGuS often operate on a *learning phase* which proposes a candidate program, and a *verification phase* which checks the proposal against the semantic restriction. To some extent, our lemma synthesis approach is similar to SyGuS since we also discover potential lemmas and verify their validity. However, we *focus on synthesizing separation logic lemmas* but not the computer programs. Our syntactic restriction is *more goal-directed* since the lemmas is controlled by specific templates, unlike the context-free-grammar restriction of SyGuS. Moreover, our semantic restriction *cannot be represented* by a SMT formula since we require that if a lemma can be proved valid, its proof must contain an induction hypothesis application. Therefore, we believe that the induction proof is *necessary* in both the lemma discovery and verification phases. This proof technique is currently *not supported* by any SyGuS-based approaches.

8 CONCLUSION

We have proposed a novel framework for synthesizing lemmas to assist in proving separation logic entailments in the fragment of symbolic-heap separation logic with inductive heap predicates and linear arithmetic. Our framework is able to synthesize various kinds of *inductive lemmas*, which help to modularize the proofs of sophisticated entailments. The synthesis of inductive lemmas is non-trivial since induction proof is required by both the lemma discovery and validation phases. In exchange, these lemmas can significantly improve the completeness of induction proof in separation logic. We have shown by experiment that our lemma-synthesis-assisted prover **SLS** is able to prove many entailments that could not be proved by the state-of-the-art separation logic provers.

We shall now discuss two limitations of our approach. Firstly, our current implementation cannot simultaneously derive new constraints from both the antecedent and the consequent of a lemma template. Theoretically, our framework can handle a lemma template with different unknown relations on both these two sides. However, the set of unknown assumptions which is introduced corresponding to these relations is far too complicated to be discharged by the current underlying prover. Secondly, we only support to infer linear arithmetic constraints with Farkas' lemma. In future, we would like to extend the lemma synthesis framework with suitable constraint solving techniques to support more kinds of pure constraints, such as sets or multisets of values.

ACKNOWLEDGMENTS

We would like to thank the reviewers of POPL'18 PC and AEC for the constructive comments on the paper and the artifact. We wish to thank Dr. Aleksandar Nanevski for his valuable suggestions on preparing the final version of this paper, and Dr. Andrew C. Myers for his dedication as the program chair of POPL'18. We are grateful for the encouraging feedback from the reviewers of OOPSLA'17 on our previous submission. The first author wish to thank Ms. Mirela Andreea Costea and Dr. Makoto Tatsuta for the inspiring discussions about the entailment proof. This research is partially supported by an NUS research grant R-252-000-553-112 and an MoE Tier-2 grant MOE2013-T2-2-146.

REFERENCES

Aws Albarghouthi, Josh Berdine, Byron Cook, and Zachary Kincaid. 2015. Spatial Interpolants. In *European Symposium on Programming (ESOP)*. 634–660.

Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. 1–25.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A Decidable Fragment of Separation Logic. In *International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. 97–109.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005a. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *International Symposium on Formal Methods for Components and Objects*. 115–137.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005b. Symbolic Execution with Separation Logic. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 52–68.

Josh Berdine, Byron Cook, and Samin Ishtiaq. 2011. SLAyer: Memory Safety for Systems-Level Code. In *International Conference on Computer Aided Verification (CAV)*. 178–183.

Marius Bozga, Radu Iosif, and Swann Perarnau. 2010. Quantitative Separation Logic and Programs with Lists. *J. Autom. Reasoning* 45, 2 (2010), 131–156.

James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. 2011. Automated Cyclic Entailment Proofs in Separation Logic. In *International Conference on Automated Deduction (CADE)*. 131–146.

James Brotherston, Carsten Fuhs, Juan A. Navarro Pérez, and Nikos Gorogiannis. 2014. A decision procedure for satisfiability in Separation Logic with inductive predicates. In *Joint Meeting of International Conference on Computer Science Logic and Symposium on Logic in Computer Science, CSL-LICS*. 25:1–25:10.

James Brotherston, Nikos Gorogiannis, Max I. Kanovich, and Reuben Rowe. 2016. Model checking for Symbolic-Heap Separation Logic with inductive predicates. In *Symposium on Principles of Programming Languages (POPL)*. 84–96.

James Brotherston and Alex Simpson. 2011. Sequent calculi for induction and infinite descent. *J. Log. Comput.* 21, 6 (2011), 1177–1216.

Alan Bundy. 2001. The Automation of Proof by Mathematical Induction. In *Handbook of Automated Reasoning (in 2 volumes)*. 845–911.

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA International Symposium on Formal Methods (NFM)*. 3–11.

Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2012. Automated verification of shape, size and bag properties via user-defined predicates in Separation Logic. *Science of Computer Programming (SCP)* 77, 9 (2012), 1006–1036.

Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. 2015. Automatic induction proofs of data-structures in imperative programs. In *Conference on Programming Language Design and Implementation (PLDI)*. 457–466.

Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *International Conference on Computer Aided Verification (CAV)*. 420–432.

Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. 2011. Tractable Reasoning in a Fragment of Separation Logic. In *International Conference on Concurrency Theory (CONCUR)*. 235–249.

Dino Distefano and Matthew J. Parkinson. 2008. jStar: towards practical verification for java. 213–226.

Constantin Enea, Ondrej Lengál, Mihaela Sighireanu, and Tomás Vojnar. 2014. Compositional Entailment Checking for a Fragment of Separation Logic. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 314–333.

Constantin Enea, Mihaela Sighireanu, and Zhilin Wu. 2015. On Automated Lemma Generation for Separation Logic with Inductive Definitions. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. 80–96.

Radu Iosif, Adam Rogalewicz, and Jiri Simácek. 2013. The Tree Width of Separation Logic with Recursive Definitions. In *International Conference on Automated Deduction (CADE)*. 21–38.

Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. 2014. Deciding Entailments in Inductive Separation Logic with Tree Automata. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. 201–218.

Quang Loc Le, Jun Sun, and Wei-Ngan Chin. 2016. Satisfiability Modulo Heap-Based Programs. In *International Conference on Computer Aided Verification (CAV)*. 382–404.

Leonardo Mendonça De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. 337–340.

Huu Hai Nguyen and Wei-Ngan Chin. 2008. Enhancing Program Verification with Lemmas. In *International Conference on Computer Aided Verification (CAV)*. 355–369.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *International Conference on Computer Science Logic (CSL)*. 1–19.

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2011. Separation Logic + Superposition Calculus = Heap Theorem Prover. In *Conference on Programming Language Design and Implementation (PLDI)*. 556–566.

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2013. Separation Logic Modulo Theories. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 90–106.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *International Conference on Computer Aided Verification (CAV)*. 773–789.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. Automating Separation Logic with Trees and Data. In *International Conference on Computer Aided Verification (CAV)*. 711–728.

Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Parthasarathy Madhusudan. 2013. Natural proofs for structure, data, and separation. In *Conference on Programming Language Design and Implementation (PLDI)*. 231–242.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Symposium on Logic in Computer Science (LICS)*. 55–74.

John C. Reynolds. 2008. An Introduction to Separation Logic. Lecture Notes for the PhD Fall School on Logics and Semantics of State, Copenhagen 2008. Retrieved on 2017, March 16th. http://www.cs.cmu.edu/~jcr/copenhagen08.pdf

Alexander Schrijver. 1986. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA.

Mihaela Sighireanu and David R. Cok. 2016. Report on SL-COMP 2014. *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2016), 173–186.

Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2016. Automated Mutual Explicit Induction Proof in Separation Logic. In *International Symposium on Formal Methods (FM)*. 659–676.

Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2017. Automated Lemma Synthesis in Symbolic-Heap Separation Logic. *Technical Report* (2017). https://arxiv.org/abs/1710.09635

Alfred North Whitehead and Bertrand Russell. 1912. *Principia Mathematica*. University Press.