Frame Inference for Inductive Entailment Proofs in Separation Logic

Quang Loc Le[1], Jun Sun[2], and Shengchao Qin[1]

[1]Teesside University, Middlesbrough, United Kingdom, {Q.Le, S.Qin}@tees.ac.uk
[2]Singapore University of Technology and Design, Singapore, sunjun@sutd.edu.sg

Abstract. Given separation logic formulae A and C, frame inference is the problem of checking whether A entails C and simultaneously inferring residual heaps. Existing approaches on frame inference do not support inductive proofs with general inductive predicates. In this work, we present an automatic frame inference approach for an expressive fragment of separation logic. We further show how to strengthen the inferred frame through predicate normalization and arithmetic inference. We have integrated our approach into an existing verification system. The experimental results show that our approach helps to establish a number of non-trivial inductive proofs which are beyond the capability of all existing tools.

1 Introduction

Separation logic (SL) [20,37] has been well established for reasoning about heap-manipulating programs (like linked-lists and trees). Often, SL is used in combination with inductive predicates to precisely specify data structures manipulated by a program. In the last decade, a large number of SL-based verification systems have been developed [1,6,3,19,8,36,33,18,29,13,24]. In these systems, SL is typically used to express assertions about program states. The problem of validating these assertions can be reduced to the *entailment* problem in SL, i.e., given two SL formulas Δ_a and Δ_c, to check whether $\Delta_a \models \Delta_c$ holds. Moreover, SL provides the frame rule [20], one prominent feature to enable compositional (a.k.a. modular) reasoning in the presence of the heap:

$$\text{FRAME RULE } \frac{\{P\}c\{Q\}}{\{P*F\}c\{Q*F\}}$$

where c is a program, P, Q and F are SL formulas, and $*$ is the separating conjunction in SL. Intuitively, $P*F$ states that P and F hold in disjoint heaps. This conjunction allows the frame rule to guarantee that F is unchanged under the action of c. This feature of SL is essential for scalability [21,44,6] as it allows the proof of a program to be decomposed (and reused) into smaller ones, e.g., proofs of procedures. To automate the application of the frame rule, SL-based proof systems rely on a generalized form of the entailment, which is referred to as frame inference [1,12,8,33,39]. That is, given Δ_a and Δ_c, to check whether Δ_a entails Δ_c and simultaneously generate the residual heap, which is a satisfiable frame Δ_f capturing properties of the memory in Δ_a that is not covered by Δ_c. This problem, especially if Δ_a and Δ_c are constituted by general

inductive predicates, is highly non-trivial as it may require inductive reasoning. Existing approaches [1,33] are limited to specific predicates e.g., linked lists and trees. The systems reported in [12,8,39] do not adequately support the frame inference problem for inductive entailments in separation logic with predicate definitions and arithmetic.

In this work, we propose a sound approach for frame inference which aims to enhance modular verification in an expressive SL fragment with general inductive predicates and Presburger arithmetic. Intuitively, given an entailment $\Delta_a \models \Delta_c$, our goal is to infer a satisfiable frame axiom Δ_f such that $\Delta_a \models \Delta_c * \Delta_f$ holds. Our approach works as follows. We first augment the entailment checking with an *unknown* second-order variable $\mathtt{U_f}(\bar{t})$ as a place-holder of the frame, where $\bar{t}$ is a set of pointer-typed variables common in Δ_a and Δ_c. That is, the entailment checking becomes $\Delta_a \models \Delta_c * \mathtt{U_f}(\bar{t})$. Afterwards, the following two steps are conducted. Firstly, we invoke a novel proof system to derive a *cyclic* proof for $\Delta_a \models \Delta_c * \mathtt{U_f}(\bar{t})$ whilst inferring a predicate which $\mathtt{U_f}$ must satisfy so that the entailment is valid. We show that the cyclic proof is valid if this predicate is satisfiable. Secondly, we strengthen the inferred frame with shape normalization and arithmetic inference.

For the first step, we design a new cyclic proof system (e.g., based on [2,3]) with an automated *cut rule* so as to effectively infer the predicate on $\mathtt{U_f}$. A cyclic proof is a derivation tree whose root is the given entailment checking and whose edges are constructed by applying SL proof rules. A derivation tree of a cyclic proof may contain virtual *back-links*, each of which links a (leaf) node back to an ancestor. Intuitively, a back-link from a node l to an internal node i means that the proof obligation at l is induced by that at i. Furthermore, to avoid potentially unsound cycles (i.e., self-cycles), a global soundness condition must be imposed upon these derivations to qualify them as genuine proofs. In this work, we develop a sequent-based cyclic proof system with a *cyclic cut* rule so as to form back-links effectively and check the soundness condition eagerly. Furthermore, we show how to extract lemmas from the proven cyclic proofs and reuse them through lemma application for an efficient proof system. These synthesized lemmas work as *dynamic cuts* in the proposed proof system.

For the second step, we strengthen the inferred predicate on the frame $\mathtt{U_f}(\bar{t})$ so that it becomes more powerful in establishing correctness of certain programs. In particular, the inferred frame is strengthened with predicate normalization and arithmetic inference. The normalization includes predicate split (i.e., to expose the spatial separation of the inferred frame) and predicate equivalence (i.e., to relate the inferred frame with user-supplied predicates). The arithmetic inference discovers predicates on pure properties (size, sum, height, content and bag) to support programs which require induction reasoning on both shape and data properties.

Lastly, we have implemented the proposal and integrated it into a modular verification engine. Our experiments show that our approach infers strong frames which enhances the verification of heap-manipulating programs.

2 Preliminaries

In this section, we present the fragment of SL which is used as the assertion language in this work. This fragment, described in Fig. 1, is expressive enough for specifying and

$$
\begin{array}{lll}
\text{Formula} & \Phi ::= \Delta \mid \Phi_1 \vee \Phi_2 & \Delta ::= \exists \bar{v} \cdot (\kappa \wedge \pi) \\
\text{Spatial} & \kappa ::= \texttt{emp} \mid x \mapsto c(\bar{v}) \mid \texttt{P}(\bar{t}) \mid \kappa_1 * \kappa_2 & \\
\text{Pure} & \pi ::= \texttt{true} \mid \texttt{false} \mid \alpha \mid \phi \mid p(\bar{t}) \mid \exists v \cdot \pi \mid \forall v \cdot \pi \mid \neg \pi \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 & \\
\text{Equality} & \alpha ::= v_1 = v_2 \mid v = \texttt{null} & \\
\text{Pres. arith.} & \phi ::= a \mid \phi_1 = \phi_2 \mid \phi_1 \leq \phi_2 & \\
 & a ::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}} \times a \mid a_1 + a_2 \mid -a \mid \mathit{max}(a_1, a_2) \mid \min(a_1, a_2) & \\
\text{Lemma} & \iota ::= P(\bar{v}) \wedge \pi \bowtie \exists \bar{w} \cdot \kappa \wedge \pi \qquad \bowtie \; ::= \; \rightarrow \mid \leftarrow \mid \leftrightarrow &
\end{array}
$$

Fig. 1. Syntax

verifying properties of a variety of data structures [24,25,41,26,35]. We use $\bar{t}$ to denote a sequence of terms and occasionally use a sequence (i.e., $\bar{t}$) to denote a set when there is no ambiguity. A formula Φ in our language is a disjunction of multiple clauses Δ, each of which is a conjunction of a spatial predicate κ and a pure (non-heap) constraint π. The spatial predicate κ captures properties of the heap whereas π captures properties of the data. κ can be an empty heap $\texttt{emp}$, or a points-to predicate $r \mapsto c(\bar{v})$ where c is a data structure, or a user-defined predicate $\texttt{P}(\bar{t})$ or a spatial conjunction $\kappa_1 * \kappa_2$. $\texttt{null}$ is a special heap location. A pure constraint π is in the form of (dis)equality α (on pointers) and Presburger arithmetic ϕ. We write $v_1 \neq v_2$ and $v \neq \texttt{null}$ for $\neg(v_1 = v_2)$ and $\neg(v = \texttt{null})$, respectively. We often omit the pure part of a formula Φ when it is $\texttt{true}$. For standardizing the notations, we use uppercase letters for unknown (to-be-inferred) predicates, (e.g., $\texttt{P}(\bar{t})$) and lowercase letters (e.g., $p(\bar{t})$) for known predicates.

A user-defined (inductive) predicate $\texttt{P}(\bar{v})$ with parameters $\bar{v}$ is defined in the form of a disjunction, i.e., $\texttt{pred}\ \texttt{P}(\bar{v}) \equiv \Phi$, where each disjunct in Φ is referred to as a branch. In each branch, variables that are not in $\bar{v}$ are implicitly existentially-quantified. We use function $\texttt{unfold}(\texttt{P}(\bar{t}))$ to replace an occurrence of inductive predicates by the disjuncts in the definition of $\texttt{P}$ with actual/formal parameters renaming. For example, the following predicates $\texttt{lseg}$ and $\texttt{lsegn}$ are defined to express list segments where every node contains the same value 1, given data structure $\texttt{node}\{\texttt{int val; node next;}\}$.

$$
\begin{array}{l}
\texttt{pred lseg}(\texttt{root}, l) \equiv \texttt{emp} \wedge \texttt{root} = l \vee \exists\, q \cdot \texttt{root} \mapsto \texttt{node}(1, q) * \texttt{lseg}(q, l); \\
\texttt{pred lsegn}(\texttt{root}, l, n) \equiv \texttt{emp} \wedge \texttt{root} = l \wedge n = 0 \vee \exists\, q \cdot \texttt{root} \mapsto \texttt{node}(1, q) * \texttt{lsegn}(q, l, n-1);
\end{array}
$$

where $\texttt{root}$ is the head, l the end of the segment and n the length of the segment.

In our framework, we may have lemmas to assist program verification. A lemma ι of the form $\Delta_l \rightarrow \Delta_r$, which means that the entailment $\Delta_l \models \Delta_r$ holds. We write $A \leftrightarrow B$, a short form of $A \rightarrow B$ and $B \rightarrow A$, to denote a two-way lemma. If $A \leftrightarrow B$, A is semantically equivalent to B. We use E and F to denote an entailment problem.

In the following, we discuss semantics of the SL fragment. Concrete heap models assume a fixed finite collection *Node*, a fixed finite collection *Fields*, a disjoint set *Loc* of locations (i.e., heap addresses), a set of non-address values *Val* such that $\texttt{null} \in \mathit{Val}$ and $\mathit{Val} \cap \mathit{Loc} = \emptyset$. The semantics is given by a satisfaction relation: $s, h \models \Phi$ that forces the stack s and heap h to satisfy the constraint Φ where $h \in \mathit{Heaps}$, $s \in \mathit{Stacks}$, and Φ is a formula. *Heaps* and *Stacks* are defined as follows.

$$
\begin{array}{l}
\mathit{Heaps} \stackrel{\text{def}}{=} \mathit{Loc} \rightharpoonup_{fin} (\mathit{Node} \times \mathit{Fields} \rightarrow \mathit{Val} \cup \mathit{Loc}) \\
\mathit{Stacks} \stackrel{\text{def}}{=} \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc}
\end{array}
$$

The details of semantics of this SL fragment follow the one in [25].

```
1 node append(node x, node y)
2   requires lln(x,i)*lln(y,j)∧i>0
3   ensures lln(res,i+j);
4 {//lln(x,i)*lln(y,j)∧i>0
5   node t=last(x);
  //ll_last(x,t,i)*lln(y,j)∧i>0
6   t->next=y;
  //α:lsegn(x,t,i−1)*t↦node(y)*lln(y,j)∧i>0
7   return x;}
```

```
8  node last(node x)
9    requires lln(x,n)∧n>0
10   ensures ll_last(x,res,n);
11 { if(!x->next) return x;
12   else return last(x->next);}
```

Fig. 2. Code of append.

3 Illustrative Example

In the following, we first discuss the limitation of the existing entailment procedures [1,8] to the frame inference problem. Given an entailment, these procedures deduce it until the following subgoal is obtained: $\Delta_a \vdash \texttt{emp} \wedge \texttt{true}$. Then, they conclude that Δ_a is the residual frame. However, these approaches provide limited support for proofs of induction. While [1] provides inference rules as a sequence of inductive reasoning for hardwired lists and trees, our previous work [8] supports inductive proofs via user-supplied lemmas [30]. Hence, it is very hard for these procedures to automatically infer the frame for the entailments which require proofs of induction.

We illustrate our approach via the verification of the append method shown in Fig. 2, which appends a singly-linked list referred to by y to the end of the singly-linked list referred to by x. It uses the auxiliary procedure last (lines 8-12) to obtain the pointer referrring to the last node in the list. Each node object x has a data value x->data and a next pointer x->next. For simplicity, we assume that every node in the x list and the y list has data value 1. The correctness of append and last is specified using our fragment of SL with a pre-condition (requires) and a post-condition (ensures). The auxiliary variable res denotes the return value of the procedure. Note that these specifications refer to the user-provided predicates lln and ll_last, which are defined as follows.

$$\texttt{pred lln}(\texttt{root},n) \equiv \texttt{emp}\wedge\texttt{root}=\texttt{null}\wedge n=0 \vee \exists\, q\cdot \texttt{root}\mapsto\texttt{node}(1,q)*\texttt{lln}(q,n-1);$$
$$\texttt{pred ll\_last}(\texttt{root},l,n) \equiv \texttt{l}\mapsto\texttt{node}(1,\texttt{null})\wedge\texttt{root}=l\wedge n=1 \vee \exists q\cdot\ \texttt{root}\mapsto\texttt{node}(1,q)*\texttt{ll\_last}(q,l,n-1);$$

Intuitively, the predicate lln(root,n) is satisfied if root points to a singly-linked list with n nodes. The predicate ll_last(t,p,n) is satisfied if t points to a list segment with last element p and length n. In our framework, we provide a library of commonly used inductive predicates (and the corresponding lemmas), including for example the definitions for list segments lseg and lsegn introduced earlier. Given these specifications, we automatically deduce predicates on the intermediate program states (using existing approaches [8]), shown as comments in Fig. 2, as well as the following three entailment checks that must be established in order to verify the absence of memory errors and the correctness of the method append.

$$\texttt{E}_1\texttt{:lln}(x,i)*\texttt{lln}(y,j)\wedge i>0 \vdash \exists\, n_1\cdot\texttt{lln}(x,n_1)\wedge n_1>0$$
$$\texttt{E}_2\texttt{:ll\_last}(x,t,i)*\texttt{lln}(y,j)\wedge i>0 \ \vdash\ \exists\, q,v\cdot t\mapsto\texttt{node}(v,q)$$
$$\texttt{E}_3\texttt{:lsegn}(\texttt{res},t,i-1)*t\mapsto\texttt{node}(1,y)*\texttt{lln}(y,j)\wedge i>0 \ \vdash\ \texttt{lln}(\texttt{res},i+j)$$

$\texttt{E}_1$ aims to establish a local specification at line 5 which we generate automatically. $\texttt{E}_2$ must be satisfied so that no null-dereference error would occur for the assignment to `t->next` at line 6. $\texttt{E}_3$ aims to establish that the postcondition is met. Frame inference is necessary in order to verify the program. In particular, frame inference for $\texttt{E}_2$ is crucial to construct a precise heap state after line 6, i.e., the state α in the figure, which is necessary to establish $\texttt{E}_3$. Furthermore, the frame of $\texttt{E}_3$ (which is inferred as `emp`) helps to show that this program does not leak memory. As the entailment checks $\texttt{E}_2$ and $\texttt{E}_3$ require both induction reasoning and frame inference, they are challenging for existing SL proof systems [12,3,8,36,31,9,15,40]. In what follows, we illustrate how our system establishes a cyclic proof with frame inference for $\texttt{E}_2$.

Frame Inference Our frame inference starts with introducing an unknown predicate (a second-order variable) $\texttt{U}_1(x,t,q,v,y)$[1] as the initial frame, which is a place-holder for a heap predicate on variables x, t, q and y (i.e., variables referred to in $\texttt{E}_2$). That is, $\texttt{E}_2$ is transformed to the following entailment checking problem:

$$\texttt{F}_2\colon \texttt{ll\_last}(x,t,i)*\texttt{lln}(y,j)\wedge i>0 \vdash_{L_0} \exists q,v\cdot t\mapsto\texttt{node}(v,q)*\texttt{U}_1(x,t,q,v,y)$$

where L_0 is a set of induction hypotheses and sound lemmas. This set is accumulated automatically during the proof search and used for constructing cyclic proofs and lemma application. If a hypothesis is proven, it becomes a lemma and may be applied latter during the proof search. in this example, initially $L_0=\emptyset$. The proposed proof system derives a cyclic proof for the entailment problem and, at the same time, infers a set of constraints $\mathcal{R}$ for $\texttt{U}_1(x,t,q,v,y)$ such that the proof is valid if the system $\mathcal{R}$ is satisfiable. Each constraint in $\mathcal{R}$ has the form of logical implication i.e., $\Delta_b \Rightarrow \texttt{U}(\bar{v})$ where Δ_b is the body and $\texttt{U}(\bar{v})$ is the head (a second-order variable). For $\texttt{F}_2$, the following two constraints are inferred, denoted by σ_1 and σ_2.

$$\begin{array}{l}\sigma_1\colon \texttt{lln}(y,j)\wedge t=x\wedge q=\texttt{null}\wedge v=1 \Rightarrow \texttt{U}_1(x,t,q,v,y)\\ \sigma_2\colon x_2\mapsto\texttt{node}(1,x)*\texttt{U}_1(x,t,q,v,y) \Rightarrow \texttt{U}_1(x_2,t,q,v,y)\end{array}$$

We then use a decision procedure (e.g., $\texttt{S2SAT}_{\texttt{SL}}$ [25,26] or [4]) to check the satisfiability of $\sigma_1\wedge\sigma_2$. Note that we write a satisfiable definition of $(\Delta_1\Rightarrow\texttt{U}(\bar{v}))\wedge(\Delta_2\Rightarrow\texttt{U}(\bar{v}))$ in the equivalent form of $\texttt{U}(\bar{v})\equiv\Delta_1\vee\Delta_2$. For instance, the above constraints are written as:

$$\begin{array}{l}\texttt{U}_1(\texttt{root},t,q,v,y) \equiv \texttt{lln}(y,j)\wedge\texttt{root}=t\wedge q=\texttt{null}\wedge v=1\\ \quad\vee \exists q_1\cdot\texttt{root}\mapsto\texttt{node}(1,q_1)*\texttt{U}_1(q_1,t,q,v,y);\end{array}$$

Note that, in the above definition of $\texttt{U}_1$, the separation of those heap-lets referred to by `root`, y and q is not explicitly captured. Additionally, relations over the sizes are also missing. Such information is necessary in order to establish the left-hand side of $\texttt{E}_3$. The successful verification of $\texttt{E}_3$ in turn establishes the postcondition of method `append`. In the following we show how to strengthen the inferred frame.

Frame Strengthening We strengthen $\texttt{U}_1$ with spatial separation constraints on the pointer variables $root$, y and q. To explicate the *spatial* separation among these pointers, our

[1] In implementation, we add # annotation into instantiated variables and non-heap variables to guide proof search which are not shown here.

system generates the following equivalent lemma and splits $\mathtt{U_1}$ into two disjoint heap regions (with $*$ conjunction):

$$\mathtt{U_1}(\mathtt{root},t,q,v,y) \equiv \mathtt{U_2}(\mathtt{root},t)*\mathtt{lln}(y,j)\wedge q=\mathtt{null}\wedge v=1$$

where $\mathtt{U_2}$ is a new auxiliary predicate with an inferred definition:

$$\mathtt{U_2}(\mathtt{root},t) \equiv \mathtt{emp}\wedge\mathtt{root}=t \vee \exists\, q_1 \cdot \mathtt{root}\mapsto\mathtt{node}(1,q_1)*\mathtt{U_2}(q_1,t)$$

Next, our system detects that $\mathtt{U_2}$ is equivalent to the user-defined predicate `lseg`, and generates the lemma: $\mathtt{U_2}(\mathtt{root},t)\leftrightarrow\mathtt{lseg}(\mathtt{root},t)$. Relating $\mathtt{U_2}$ to `lseg` enhances the understanding of the inferred predicates. Furthermore, as shown in [9], this relation helps to reduce the requirements of induction reasoning among equivalent inductive predicates with different names. Substituting $\mathtt{U_2}$ with the equivalent `lseg`, $\mathtt{U_1}$ becomes:

$$\mathtt{U_1}(\mathtt{root},t,q,v,y) \equiv \mathtt{lseg}(\mathtt{root},t)*\mathtt{lln}(y,j)\wedge q=\mathtt{null}\wedge v=1$$

This definition states that frame $\mathtt{U_1}$ holds in two disjoint heaps: one list segment pointed to by `root` and a list pointed to by y. After substitution the entailment $\mathtt{F_2}$ becomes

$$\mathtt{ll\_last}(x,t,i)*\mathtt{lln}(y,j)\wedge i>0 \vdash_{L_0} \mathtt{t}\mapsto\mathtt{node}(1,\mathtt{null})*\mathtt{lseg}(x,t)*\mathtt{lln}(y,j)$$

Next, we further strengthen the frame with pure properties, which is necessary to successfully establish the left hand side of $\mathtt{E_3}$. In particular, we generate constraints to capture that the numbers of allocated heaps in the left hand side and the right hand side of $\mathtt{F_2}$ are identical. Our system obtains these constraints through two phases. First, it automatically augments an argument for each inductive predicate in $\mathtt{F_2}$ to capture its size property. Concretely, it detects that while predicates `ll_last` and `lln` have such size argument already, the shape-based frame `lseg` has not. As so, it extends $\mathtt{lseg}(\mathtt{root},t)$ to obtain the predicate $\mathtt{lsegn}(\mathtt{root},t,m)$ where the size property is captured by parameter m. Now, we substitute the `lsegn` into $\mathtt{F_2}$ to obtain:

$$\mathtt{ll\_last}(x,t,i)*\mathtt{lln}(y,j)\wedge i>0 \vdash_{L_0} \exists k\cdot t\mapsto\mathtt{node}(1,\mathtt{null})*\mathtt{lsegn}(x,t,k)*\mathtt{lln}(y,j)$$

After that, we apply the same three steps of frame inference to generate the size constraint: constructing unknown predicates, proving entailment and inferring a set of constraints and checking satisfiability. For the first step, the above entailment is enriched with one unknown (pure) predicate: $\mathtt{P_1}(i,j,k)$ which is the place-holder for arithmetical constraints among size variables i, j and k. The augmented entailment checking is:

$$\begin{array}{l}\mathtt{ll\_last}(x,t,i)*\mathtt{lln}(y,j)\wedge i>0\\ \quad\vdash_{L_0} \exists k\cdot\mathtt{lsegn}(x,t,k)*t\mapsto\mathtt{node}(1,\mathtt{null})*\mathtt{lln}(y,j)\wedge\mathtt{P_1}(i,j,k)\end{array}$$

Secondly, our system successfully derives a proof for the above entailment under condition that the following disjunctive set of two constraints is satisfiable.

$$\begin{array}{l}\sigma_3\colon\ i=1\wedge k=0 \Rightarrow \mathtt{P_1}(i,j,k)\\ \sigma_4\colon\ i_1=i-1\wedge k_1=k-1\wedge i>0\wedge\mathtt{P_1}(i_1,j,k_1) \Rightarrow \mathtt{P_1}(i,j,k)\end{array}$$

Lastly, to check whether the $\sigma_3\wedge\sigma_4$ is satisfiable, we automatically compute the closure form for $\sigma_3\wedge\sigma_4$ as: $\mathtt{P_1}(i,j,k) \equiv k=i-1\wedge i>0$. This formula is satisfiable and substituted into the frame as: $\mathtt{lsegn}(x,t,k)*\mathtt{lln}(y,j)\wedge q=\mathtt{null}\wedge v=1\wedge k=i-1\wedge i>0$.

$$\text{EMP}\ \frac{\pi_a \implies \pi_c}{\texttt{emp}\wedge\pi_a \vdash_L \texttt{emp}\wedge\pi_c \rightsquigarrow \texttt{true}} \qquad \text{CCUT}\ \frac{\begin{array}{c}\Delta_l \rightarrow \Delta_r \in L \quad \Delta_{a_1}\rho \vdash_L \Delta_l \rightsquigarrow \mathcal{R}_1 \\ \Delta_r\rho * \Delta_{a_2} \vdash_L \Delta_c \rightsquigarrow \mathcal{R}_2\end{array}}{\Delta_{a_1} * \Delta_{a_2} \vdash_L \Delta_c \rightsquigarrow \mathcal{R}_1 \wedge \mathcal{R}_2}\ \text{gsc}$$

$$\text{PRED-M}\ \frac{\Delta_a \vdash_L \Delta_c \wedge \text{freeEQ}([\bar{v}/\bar{w}]) \rightsquigarrow \mathcal{R}}{\texttt{P}(r,\bar{v}) * \Delta_a \vdash_L \texttt{P}(r,\bar{w}) * \Delta_c \rightsquigarrow \mathcal{R}} \qquad \text{M}\ \frac{\Delta_a \vdash_L \Delta_c \wedge \text{freeEQ}([\bar{v}/\bar{w}]) \rightsquigarrow \mathcal{R}}{x \mapsto c(\bar{v}) * \Delta_a \vdash_L x \mapsto c(\bar{w}) * \Delta_c \rightsquigarrow \mathcal{R}}$$

$$\text{RU}\ \frac{\begin{array}{c}(x \mapsto c(\bar{w}) \text{ or } x = \texttt{null} \text{ or } x = q) \in \Delta_a \\ \texttt{unfold}(\texttt{P}(x,\bar{v})) = \{\Delta_1, .., \Delta_n\} \quad \exists i \in \{1...n\} \cdot \Delta_a \vdash_L \Delta_i * \Delta_c \rightsquigarrow \mathcal{R}_i\end{array}}{\Delta_a \vdash_L \texttt{P}(x,\bar{v}) * \Delta_c \rightsquigarrow \mathcal{R}_i}$$

$$\text{LU}\ \frac{\begin{array}{c}(x \mapsto c(\bar{w}) \text{ or } x = \texttt{null} \text{ or } x = q) \in \Delta_c \quad \texttt{unfold}(\texttt{P}(x,\bar{v})) = \{\Delta_1, .., \Delta_n\} \\ L_i' = L \cup \{(\texttt{P}(x,\bar{v}) * \Delta_a \rightarrow \Delta_c)\} \quad \forall i \in \{1...n\} \cdot \Delta_i * \Delta_a \vdash_{L_i'} \Delta_c \rightsquigarrow \mathcal{R}_i\end{array}}{\texttt{P}(x,\bar{v}) * \Delta_a \vdash_L \Delta_c \rightsquigarrow \bigwedge \mathcal{R}_i}$$

Fig. 3. Basic Inference Rules for Entailment Procedure (where gsc is global soundness condition)

4 Frame Inference

In this section, we present our approach for frame inference in detail. Given an entailment $\Delta_a \vdash \Delta_c$, where Δ_a is the antecedent (LHS) and Δ_c is the consequence (RHS), our system attempts to infer a frame Δ_f such that when a frame is successfully inferred, the validity of the entailment $\Delta_a \vdash \Delta_c * \Delta_f$ is established at the same time.

Our approach has three main steps. Firstly, we enrich RHS with an unknown predicate in the form of $\texttt{U}(\bar{v})$ to form the entailment $\Delta_a \vdash_L \Delta_c * \texttt{U}(\bar{v})$ where $\bar{v}$ includes all free pointer-typed variables of Δ_a and Δ_c and L is the union of a set of user-supplied lemmas and a set of induction hypotheses (initially $\emptyset$). Among these, the parameters are annotated with $\#$ following the principle that instantiation (and subtraction) must be done before inference. The detail is as follows: (i) all common variables of Δ_a and Δ_c are $\#$-annotated; (ii) points-to pointers of Δ_c are $\#$-annotated; (iii) the remaining pointers are not $\#$-annotated. In the implementation, inference of frame predicates is performed incrementally such that shape predicates are inferred prior to pure ones. Secondly, we construct a proof of the entailment and infer a set of constraints $\mathcal{R}$ for $\texttt{U}(\bar{v})$.Thirdly, we check the satisfiability of $\mathcal{R}$ using the decision procedure in [25,26].

In the following, we present our entailment checking procedure with a set of proof rules shown in Fig. 3 and 4. For each rule, the obligation is at the bottom and its reduced form is on the top. In particular, the rules in Fig. 3 are used for entailment proving (i.e., to establish a cyclic proof) and the rules in Fig. 4 are used for predicate inference.

Given an entailment check in the form of $\Delta_a \vdash_L \Delta_c$, the rules shown in Fig. 3 are designed to subtract the heap (via the rules $[\underline{\text{M}}]$ and $[\underline{\textbf{PRED}-\textbf{M}}]$) on both sides until their heaps are empty. After that, it checks the validity for the implication of two pure formulas by using an SMT solver, like Z3 [27], as shown in rule $[\underline{\textbf{EMP}}]$. Algorithmically, this entailment checking is performed as follows.

– **Matching.** The rules $[\underline{\text{M}}]$ and $[\underline{\textbf{PRED}-\textbf{M}}]$ are used to match up identified heap chains. Starting from identified root pointers, the procedure keeps matching all their reach-

$$\text{AF}\ \frac{\sigma \equiv \mathtt{R}(r,\bar{t})*\mathtt{U_i}(r_i,\bar{t}_i\#)*\mathtt{U_f}(\bar{w},\bar{t}',\bar{z}\#,r\#)\wedge\bigtriangledown(\bar{w}\cup\{r,r_i\},\pi_1)\Rightarrow \mathtt{U}(r,\bar{w},\bar{z}\#) \quad r_i\in\bar{t} \quad \bar{t}'{=}\bar{t}\backslash(\bar{w}\cup\bar{z}\cup\{r\}) \quad \kappa_1\wedge\pi_1\vdash_L \mathtt{U_f}(\bar{w},\bar{t}',\bar{z}\#,r\#)*\kappa_2\wedge\pi_2\rightsquigarrow\mathcal{R}}{\mathtt{R}(r,\bar{t})*\mathtt{U_i}(r_i,\bar{t}_i\#)*\kappa_1\wedge\pi_1\vdash_L \mathtt{U}(r,\bar{w},\bar{z}\#)*\kappa_2\wedge\pi_2\rightsquigarrow\sigma\wedge\mathcal{R}}$$

$$\text{AF-F}\ \frac{\sigma \equiv \mathtt{U}(r,\bar{w},\bar{z}\#)\Rightarrow \mathtt{R}(r,\bar{t})*\mathtt{U_i}(r_i,\bar{t}_i\#)*\mathtt{U_f}(\bar{w},\bar{t}',\bar{z}\#,r\#)\wedge\bigtriangledown(\bar{w}\cup\{r,r_i\},\pi_1) \quad r_i\in\bar{t} \quad \bar{t}'{=}\bar{t}\backslash(\bar{w}\cup\bar{z}\cup\{r\}) \quad \mathtt{U_f}(\bar{w},\bar{t}',\bar{z}\#,r\#)*\kappa_1\wedge\pi_1\vdash_L \kappa_2\wedge\pi_2\rightsquigarrow\mathcal{R}}{\mathtt{U}(r,\bar{w},\bar{z}\#)*\kappa_1\wedge\pi_1\vdash_L \mathtt{R}(r,\bar{t})*\mathtt{U_i}(r_i,\bar{t}_i\#)*\kappa_2\wedge\pi_2\rightsquigarrow\sigma\wedge\mathcal{R}}$$

Fig. 4. Inference Rules with Predicate Synthesis.

able heaps. It unifies corresponding fields of matched roots by using the following auxiliary function freeEQ(ρ): freeEQ($[u_i/v_i]_{i=1}^n$) = $\bigwedge_{i=1}^n\{u_i = v_i\}$.

- **Unfolding.** The rules [$\underline{\mathbf{LU}}$] and [$\underline{\mathbf{RU}}$] are used to derive alternative heap chains. While rule [$\underline{\mathbf{LU}}$] presents the unfolding in the antecedent, [$\underline{\mathbf{RU}}$] in the consequent.
- **Applying Lemma.** Rule [$\underline{\mathbf{CCUT}}$] derives yet other alternative heap chains. For LHS which has at least one UD predicate, we attempt to apply a lemma as an alternative search using [$\underline{\mathbf{CCUT}}$] rule. We notice that as we assume that a lemma which is supplied by the user is valid, applying this lemma does not requires the global condition.

Cyclic Proof The proof rules in Fig. 3 are designed to establish cyclic proofs. In the following, we briefly describe a cyclic proof technique enhancing the proposal in [2].

Definition 1 (Pre-proof) *A* pre-proof *of entailment E is a pair ($\mathcal{T}_i$, $\mathcal{L}$) where $\mathcal{T}_i$ is a derivation tree and $\mathcal{L}$ is a back-link function such that: the root of $\mathcal{T}_i$ is E; for every edge from E_i to E_j in $\mathcal{T}_i$, E_i is a conclusion of an inference rule with a premise E_j. There is a back-link between E_c and E_l if there exists $\mathcal{L}(E_l) = E_c$ (i.e., $E_c = E_l\,\theta$ with some substitution θ) ; and for every leaf E_l, E_l is an axiom rule (without conclusion).*

If $\mathcal{L}(E_l) = E_c$, E_l (resp. E_c) is referred as a bud (resp. companion).

Definition 2 (Trace) *Let ($\mathcal{T}_i$, $\mathcal{L}$) be a pre-proof of $\Delta_a\vdash_L\Delta_c$; $(\Delta_{a_i}\vdash_{L_i}\Delta_{c_i})_{i\geq 0}$ be a path of $\mathcal{T}_i$. A trace following $(\Delta_{a_i}\vdash_{L_i}\Delta_{c_i})_{i\geq 0}$ is a sequence $(\alpha_i)_{i\geq 0}$ such that each α_i (for all $i{\geq}0$) is an instance of the predicate $\mathtt{P}(\bar{t})$ in the formula Δ_{a_i}, and either:*

- *α_{i+1} is the subformula containing an instance of $\mathtt{P}(\bar{t})$ in $\Delta_{a_{i+1}}$;*
- *or $\Delta_{a_i}\vdash_{L_i}\Delta_{c_i}$ is the conclusion of an unfolding rule, α_i is an instance predicate $\mathtt{P}(\bar{t})$ in Δ_{a_i} and α_{i+1} is a subformula $\Delta[\bar{t}/\bar{v}]$ which is a definition rule of the inductive predicate $\mathtt{P}(\bar{v})$. i is a progressing point of the trace.*

To ensure that a pre-proof is sound, a global *soundness condition* must be imposed to guarantee well-foundedness.

Definition 3 (Cyclic proof) *A pre-proof ($\mathcal{T}_i$, $\mathcal{L}$) of $\Delta_a\vdash_L\Delta_c$ is a cyclic proof if, for every infinite path $(\Delta_{a_i}\vdash_{L_i}\Delta_{c_i})_{i\geq 0}$ of $\mathcal{T}_i$, there is a tail of the path $p{=}(\Delta_{a_i}\vdash_{L_i}\Delta_{c_i})_{i\geq n}$ such that there is a trace following p which has infinitely progressing points.*

Brotherston *et. al.* proved [2] that $\Delta_a \vdash \Delta_c$ holds if there is a cyclic proof of $\Delta_a \vdash_\emptyset \Delta_c$ where Δ_a and Δ_c do not contain any unknown predicate.

In the following, we explain how cyclic proofs are constructed using the proof rules shown in Fig. 3. [LU] and [CCUT] are the most important rules for forming back-links and then pre-proof construction. While rule [LU] accumulates possible companions and stores them in historical sequents L, [CCUT] links a bud with a companion using some substitutions as well as checks the global soundness condition eagerly. Different to the original cyclic system [3], our linking back function only considers companions selected in the set of historical sequents L. Particularly, $\Delta_l \rightarrow \Delta_r \in L$ is used as an intelligent cut as follows. During proof search, a subgoal (i.e., $\Delta_{a_1} * \Delta_{a_2} \vdash_L \Delta_c$) may be matched with the above historical sequent to form a cycle and close the proof branch using the following principle. First, $\Delta_l \vdash \Delta_r$ is used as an induction hypothesis. As so, we have $\Delta_l \rho * \Delta_{a_2} \models \Delta_r \rho * \Delta_{a_2}$ where ρ are substitutions including those for avoiding clashing of variables between Δ_r and Δ_{a_2}. If both $\Delta_{a_1} * \Delta_{a_2} \vdash_L \Delta_l \rho * \Delta_{a_2}$ and $\Delta_r \rho * \Delta_{a_2} \vdash_L \Delta_c$ are proven, then we have:

$$\Delta_{a_1} * \Delta_{a_2} \implies \Delta_l \rho * \Delta_{a_2} \implies \Delta_r \rho * \Delta_{a_2} \implies \Delta_c.$$

Thus, the subgoal $\Delta_{a_1} * \Delta_{a_2} \vdash_L \Delta_c$ holds. We remark that if a hypothesis is proven, it can be applied as a valid lemma subsequently.

In our system, often a lemma includes universally quantified variables. We thus show a new mechanism to instantiate those lemmas that include universally quantified variables. We denote constraints with universal variables as universal guards $\forall G$. A universal guard $\forall G$ is equivalent to an *infinite* conjunction $\bigwedge_\rho G[\rho]$. Linking a leaf with universal guards is not straightforward. For illustration, let us consider the following bud $\mathtt{B}_0$ and the universally quantified companion/lemma $\mathtt{C}_0 \in L$.

$$\begin{array}{l}\mathtt{B}_0{:}\mathtt{lsegn}(\mathtt{root},\mathtt{null},n) \wedge n{=}10 \vdash_L \exists r{\cdot}\mathtt{lsegn}(\mathtt{root},r,3) * \mathtt{lsegn}(r,\mathtt{null},7)\\ \mathtt{C}_0{:}\forall a,b{\cdot}\mathtt{lsegn}(\mathtt{root},\mathtt{null},n) \wedge n{=}a{+}b \wedge a{\geq}0 \wedge b{\geq}0\\ \qquad \rightarrow \exists r{\cdot}\mathtt{lsegn}(\mathtt{root},r,a) * \mathtt{lsegn}(r,\mathtt{null},b)\end{array}$$

As shown in rule [CCUT], to link $\mathtt{B}_0$ back to $\mathtt{C}_0$, the LHS of these two entailments must be implied through some substitution. To obtain that, we propose *lemma instantiation*, a sound solution for universal lemma application. Based on the constraints in the LHS of the bud, our technique instantiates a universally quantified guard (of the selected companion/lemma) before linking it back. Concretely, we replace the universal guard by a *finite* set of its instances; an instantiation of a formula $\forall \bar{v} G(\bar{t})$ is $G(\bar{t})[\bar{w}/\bar{v}]$ for some vector of terms $\bar{w}$. These instances are introduced based on the instantiations in both LHS and RHS of the corresponding bud e.g., $n{=}10 \wedge a{=}3 \wedge b{=}7$ in $\mathtt{B}_0$.

Frame Inference The two inference rules shown in Fig. 4 are designed specifically to infer constraints for frame. In these rules, $\bigtriangledown(\bar{w}, \pi)$ is an auxiliary function that existentially quantifies free variables in π that are not in the set $\bar{w}$. This function extracts relevant arithmetic constraints to define data contents of the unknown predicates. $\mathtt{R}(r,\bar{t})$ is either $r{\mapsto}c(\bar{t})$ or a known (defined) predicate $\mathtt{P}(r,\bar{t})$, or an unknown predicate $\mathtt{U}'(r,\bar{t},\bar{w}_\#)$. The # in the unknown predicates is used to guide inference and proof search. We only infer on pointers without $_\#$-annotation. $\mathtt{U}_\mathtt{f}(\bar{w}, \bar{t'})$ is another unknown

predicate which is generated to infer the shape of pointers $\bar{w}$. Inferred pointers are annotated with $\#$ to avoid double inference. A new unknown predicate $\mathtt{U_f}$ is generated only if there exists at least one parameter not to be annotated with $\#$ (i.e., $\bar{w} \cup \bar{t}' \neq \emptyset$). To avoid conflict between the inference rules and the other rules (e.g., unfolding and matching), root pointers of a heap formula must be annotated with $\#$ in unknown predicates. For example, in our system while $x \mapsto \mathtt{c_1}(y) * \mathtt{U_1}(x_{\#}, y)$ is legal, $x \mapsto \mathtt{c_1}(y) * \mathtt{U_1}(x, y)$ is illegal. Our system applies subtraction on the heap pointed to by x rather than inference for the following check: $x \mapsto \mathtt{c_1}(\mathtt{null}) \vdash_L x \mapsto \mathtt{c_1}(y) * \mathtt{U_1}(x_{\#}, y)$.

Soundness The soundness of the inference rules in Fig. 3 has been shown in unfold-and-match systems for general inductive predicates [3,8]. In the following, we present the soundness of the inference rules in Fig. 4. We introduce the notation $\mathcal{R}(\Gamma)$ to denote a set of predicate definitions $\Gamma = \{\mathtt{U_1}(\bar{v}_1) \equiv \Phi_1, ..\mathtt{U}_n(\bar{v}_n) \equiv \Phi_n\}$ satisfying the set of constraints $\mathcal{R}$. That is, for all constraints $\Delta_l \Rightarrow \Delta_r \in \mathcal{R}$, (i) Γ contains states (s_i, h_i), a predicate definition for each unknown predicate appearing in Δ_l and Δ_r; (ii) by interpreting all unknown predicates according to Γ, then it is provable that Δ_l implies Δ_r (i.e., there exists $s_i \subseteq s$, $h_i \subseteq h$ for $i \in \{1..n\}$, and $s, h \models \Delta_l$ implies $s, h \models \Delta_r$), written as $\Gamma : \Delta_l \vdash \Delta_r$.

Lemma 1. *Given the entailment judgement $\Delta_a \vdash_{\{\}} \Delta_c \rightsquigarrow \mathcal{R}$, if there is Γ such that $\mathcal{R}(\Gamma)$, the entailment $\Gamma{:}\Delta_a \vdash \Delta_c$ holds.*

The soundness of the predicate synthesis requires that if definitions generated for unknown predicates are satisfiable, then the entailment is valid.

Theorem 1. *Given the entailment judgement $\Delta_a \vdash_{\emptyset} \Delta_c \rightsquigarrow \mathcal{R}$ $\Delta_a(\Gamma) \vdash \Delta_c(\Gamma)$ holds if there exists a solution Γ of $\mathcal{R}$.*

Theorem 1 follows from the soundness of the rules in Fig. 3 and Lemma 1.

5 Extensions

In this section, we present two ways to strengthen the inferred frame, by inferring pure properties and by normalizing inductive predicates.

Pure Constraint Inference The inferred frame is strengthened with pure constraints following two phases. We first enrich the shape-base frame with pure properties such as size, height, sum, set of addresses/values, and their combinations. After that, we apply the same three steps in section 4 to infer relational assumptions on the new pure properties. Lastly, we check satisfiability of these assumptions using FixCalc [34].

In the following, we describe how to infer size properties given a set of dependent predicates. We can similarly infer properties on height, set of addresses and values properties. We first extend an inductive predicate with a size function to capture size properties. That is, given an inductive predicate $\mathtt{P}(\bar{v}) \equiv \bigvee \Delta_i$, we generate a new predicate $\mathtt{Pn}$ with a new size parameter n as: $\mathtt{Pn}(\bar{v}, n) \equiv \bigvee(\Delta_i \wedge n = sizeF(\Delta_i))$ where function $sizeF$ is inductively defined as follows.

$$sizeF(r \mapsto c(\bar{t})) = 1 \quad sizeF(\exists \bar{v} \cdot \kappa \wedge \pi) = sizeF(\kappa)$$
$$sizeF(\mathtt{emp}) = 0 \quad sizeF(\kappa_1 * \kappa_2) = sizeF(\kappa_1) + sizeF(\kappa_2)$$
$$sizeF(\mathtt{P}(\bar{t})) = t_s \text{ where } t_s \in \bar{t} \text{ and } t_s \text{ is a size parameter}$$

To support pure properties, we extend the proposed cyclic proof system with bi-abduction for pure constraints which was presented in [43]. In particular, we adopt the abduction rules to generate relational assumptions over the pure properties in LHS and RHS. These rules are applied exhaustively until no more unknown predicates occur.

Normalization We aim to relate the inferred frame to existing user-provided predicates if possible as well as to explicate the heap separation (a.k.a. pointer non-aliasing) which may be implicitly constrained through predicates. Particularly, we present a lemma synthesis mechanism to explore relations between inductive predicates. Our system processes each inductive predicate in four steps. First, it generates *heap-only* conjectures (with quantifiers). Secondly, it enriches these conjectures with unknown predicates. Thirdly, it invokes the proposed entailment procedure to prove these conjectures, infer definitions for the unknown predicates and synthesize the lemmas. Last, it strengthens the inferred lemma with pure inference.

In the following, we present two types of normalization. This first type is to generate equivalence lemmas. This normalization equivalently matches a new generated predicate to an existing predicate in a given predicate library. Under the assumption that a library of predicates is provided together with advanced knowledge (i.e., lemmas in [1]) to enhance completeness. This normalization helps to reuse this knowledge for the new synthesized predicates, and potentially enhance the completeness of the proof system. Intuitively, given a set $\mathtt{S}$ of inductive predicates and another inductive predicate $\mathtt{P}$ (which is not in $\mathtt{S}$), we identify all predicates in $\mathtt{S}$ which are equivalent to $\mathtt{P}$. Heap-only conjecture is generated to explore the equivalent relation between two predicates, e.g., in the case of $\mathtt{P}(x, \bar{v})$ and $\mathtt{Q}(x, \bar{w})$: $\forall \bar{v} \cdot \mathtt{P}(\mathtt{root}, \bar{v}) \rightarrow \exists \bar{w} \cdot \mathtt{Q}(\mathtt{root}, \bar{w})$. The shared root parameter x has been identified by examining all permutations of root parameters of the two predicates. Moreover, our system synthesizes lemmas incrementally for the combined domains of shape and pure properties. For example, with `lln` and `lsegn`, our system generates the following lemma afterwards: $\mathtt{lsegn}(\mathtt{root},\mathtt{null},n) \leftrightarrow \mathtt{lln}(\mathtt{root},n)$.

The other type of normalization is to generate separating lemmas. This normalization aims to expose hidden separation of heaps in inductive definitions. This paragraph explores parallel or consequence separate relations over inductive predicates parameters. Two parameters of a predicate are *parallel* separating if they are both root parameters e.g., r_1 and r_2 of the predicate `zip2` as follows.

$$\begin{array}{l}\mathtt{zip2}(r_1,r_2,n) \equiv \mathtt{emp} \wedge r_1 = \mathtt{null} \wedge r_2 = \mathtt{null} \wedge n = 0 \\ \quad \vee\ r_1 \mapsto c_1(q_1) * r_2 \mapsto c_1(q_2) * \mathtt{zip2}(q_1,q_2,n-2);\end{array}$$

Two arguments of a predicate are *consequence* separating if one is a root parameter and another is reachable from the root in all base formulas derived by unfolding the predicate (e.g., those of the predicate `ll_last`). We generate these separating lemmas to explicate separation globally. As a result, the separation of actual parameters is externally visible to analyses. This visible separation enables strong updates in a modular heap analysis or frame inference in modular verification. Suppose r_1, r_2 are consequence or parallel parameters in $\mathtt{Q}(r_1, r_2, \bar{w})$, heap conjecture is generated as:

$$\mathtt{Q}(r_1, r_2, \bar{w}) \rightarrow \mathtt{Q}_1(r_1) * \mathtt{Q}_2(r_2) * \mathtt{Q}_3(\bar{w})$$

This technique could be applied to synthesize spit/join lemmas to transform predicates into the fragment of linearly compositional predicates [15,14]. For example, our system splits the predicate zip2 into two separating singly-lined lists through the following equivalent lemma: $\texttt{zip2}(\texttt{root},r_2,n) \leftrightarrow \texttt{lln}(\texttt{root},n)*\texttt{lln}(r_2,n)$.

6 Implementation and Experiments

We have implemented the proposed ideas into a procedure called S2ENT for entailment checking and frame inference, based on the SLEEK [8]. S2ENT relies on the SMT solver Z3 [27] to check satisfiability of arithmetical formulas. We have also integrated S2ENT into the verifier S2 [24].We have conducted two sets of experiments to evaluate the effectiveness and efficiency of S2ENT. The first set of experiments are conducted on a set of inductive entailment checking problems gathered from previous publications [1,5,9]. We compare S2ENT with the state-of-the-art tools to see how many of these problems can be solved. In the second set of experiments, we apply S2ENT to conduct modular verification of a set of non-trivial programs. The experiments are conducted on a machine with the Intel i3-M370 (2.4GHz) processor and 3 GB of RAM.

Entailment Proving In Table 1, we evaluate S2ENT on a set of 36 *valid* entailment problems that require induction reasoning techniques. In particular, Ent 1-5 were taken from Smallfoot [1], Ent 6-19 from $\texttt{Cyclic}_{SL}$ [3,5], Ent 20-28 from [9], and Ent 29-36 were generated by us. We evaluate S2ENT against the existing proof systems presented for user-defined predicates. While the tools reported in [12,8,36] could handle a subset of these benchmarks if users provide auxiliary lemmas/axioms, [15] was designed neither for those inductive predicates in Ent 6-28 nor frame problems in Ent 29-36. The only two tools which we can compare S2ENT with are $\texttt{Cyclic}_{SL}$ [3] and songbird [40].

The experimental results are presented in Table 1. The second column shows the entailment problems. Column bl captures the number of back-links in cyclic proofs generated by S2ENT. We observe that most of problems require only one back-link in the cyclic proofs, except that Ent 4 requires two back-links and Ent 13-15 of mutual inductive odd-even singly linked lists require three back-links. The last three columns show the results of $\texttt{Cyclic}_{SL}$, songbird and S2ENT respectively. Each cell shown in these columns is either CPU times (in seconds) if the tool proves successfully, or TO if the tool runs longer than 30s, or X if the tool returns a false positive, or NA if the entailment is beyond the capability of the tool. *In summary, out of the 36 problems,* $\texttt{Cyclic}_{SL}$ *solves 18 (with one TO - Ent 4);* songbird *solves 25 (with two false positive - Ent 17 and 27 and one TO - Ent 23); and* S2ENT *solves all 36 problems.*

In Table 1, each entailment check in Ent 1-19 has emp as frame axioms (their LHS and RHS have the same heaps). Hence, they may be handled by existing inductive proof systems like [3,9,15,40]. In particular, Ent 1-19 include shape-only predicates. The results show that $\texttt{Cyclic}_{SL}$ and songbird ran a bit faster than S2ENT in most of the their successful cases. It is expected as S2ENT requires additional steps for frame inference. Each entailment check in Ent 20-28 includes inductive predicates with pure properties (e.g., size and sortedness). While $\texttt{Cyclic}_{SL}$ can provide inductive reasoning for arithmetic and heap domains separately [5], there is no system proposed for cyclic

Table 1. Inductive Entailment Checks

Ent	Proven	bl	[3]	[40]	Ours
1	$\texttt{lseg}(x,t)*\texttt{lseg}(t,\texttt{null}) \vdash \texttt{lseg}(x,\texttt{null})$	1	0.03	0.04	0.06
2	$\texttt{lseg}(x,t)*\texttt{t}\mapsto c_1(y)*\texttt{lseg}(y,\texttt{null}) \vdash \texttt{lseg}(x,\texttt{null})$	1	0.03	0.04	0.08
3	$\texttt{lseg}(x,t)*\texttt{lseg}(t,y)*\texttt{y}\mapsto c_1(\texttt{null}) \vdash \texttt{lseg}(x,\texttt{null})$	1	0.03	1.36	0.11
4	$\texttt{lseg}(x,t)*\texttt{lseg}(t,y)*\texttt{bt}(y)\wedge y\neq\texttt{null} \vdash \texttt{lseg}(x,y)*\texttt{bt}(y)$	2	TO	0.12	0.21
5	$\texttt{lseg}(x,t)*\texttt{lseg}(t,y)*\texttt{lseg}(y,z)\wedge y\neq z \vdash \texttt{lseg}(x,y)*\texttt{lseg}(y,z)$	1	3.00	0.48	0.57
6	$\texttt{x}\mapsto c_1(y)*\texttt{rlseg}(y,z) \vdash \texttt{rlseg}(x,z)$	1	0.02	0.04	0.10
7	$\texttt{nlseg}(x,z)*\texttt{z}\mapsto c_1(y) \vdash \texttt{nlseg}(x,y)$	1	0.02	0.04	0.04
8	$\texttt{nlseg}(x,z)*\texttt{nlseg}(z,y) \vdash \texttt{nlseg}(x,y)$	1	0.03	0.04	0.06
9	$\texttt{glseg}(x,z)*\texttt{z}\mapsto c_1(y) \vdash \texttt{glseg}(x,y)$	1	0.02	0.04	0.04
10	$\texttt{glseg}(x,z)*\texttt{glseg}(z,y) \vdash \texttt{glseg}(x,y)$	1	0.02	0.04	0.04
11	$\texttt{dlseg}(u,v,x,y) \vdash \texttt{glseg}_2(u,v)$	1	0.07	0.04	0.04
12	$\texttt{dlseg}(w,v,x,z)*\texttt{dlseg}(u,w,z,y) \vdash \texttt{dlseg}(u,v,x,y)$	1	0.04	0.03	0.11
13	$\texttt{listo}(x,z)*\texttt{listo}(z,\texttt{null}) \vdash \texttt{liste}(x,\texttt{null})$	3	0.06	0.04	0.06
14	$\texttt{liste}(x,z)*\texttt{liste}(z,\texttt{null}) \vdash \texttt{liste}(x,\texttt{null})$	3	0.18	0.04	0.12
15	$\texttt{listo}(x,z)*\texttt{liste}(z,y) \vdash \texttt{listo}(x,y)$	3	4.33	0.03	0.88
16	$\texttt{binPath}(x,z)*\texttt{binPath}(z,y) \vdash \texttt{binPath}(x,y)$	1	0.03	0.05	0.06
17	$\texttt{binPath}(x,y) \vdash \texttt{binTreeSeg}(x,y)$	1	0.12	X	0.08
18	$\texttt{binTreeSeg}(x,z)*\texttt{binTreeSeg}(z,y) \vdash \texttt{binTreeSeg}(x,y)$	1	0.20	0.07	0.66
19	$\texttt{binTreeSeg}(x,y)*\texttt{binTree}(y) \vdash \texttt{binTree}(x)$	1	0.06	0.05	0.03
20	$\texttt{tmp}(x,size) \vdash \exists y\cdot \texttt{ls}(x,y,size)$	1	NA	0.73	0.39
21	$\texttt{sortll}(x,min) \vdash \texttt{ll}(x)$	1	NA	0.1	0.05
22	$\texttt{sortlln}(x,min,size) \vdash \texttt{lln}(x,size)$	1	NA	0.55	0.12
23	$\texttt{sortlln}(x,min,size) \vdash \texttt{sortll}(x,min)$	1	NA	TO	0.08
24	$\texttt{lsegn}(x,y,sz_1)*\texttt{lsegn}(y,z,sz_2) \vdash \texttt{lsegn}(x,z,sz_1+sz_2)$	1	NA	10.49	0.12
25	$\texttt{lsegn}(x,y,size_1)*\texttt{lsn}(y,size_2) \vdash \texttt{lsn}(x,size_1+size_2)$	1	NA	10.59	0.10
26	$\texttt{lsegn}(x,tl,n_1)*\texttt{tl}\mapsto c_1(y)*\texttt{lseg1}(y,ty,n_2) \vdash \texttt{lseg1}(x,ty,n_1+n_2+1)$	1	NA	16.76	0.10
27	$\texttt{avl}(x,size,height,bal) \vdash \texttt{btn}(x,size)$	1	NA	X	0.08
28	$\texttt{tll}(x,ll,lr,size) \vdash \texttt{btn}(x,size)$	1	NA	0.13	0.07
29	$\texttt{ll\_last}(x,y) \vdash y\mapsto c_1(\texttt{null})$	1	NA	NA	0.18
30	$\texttt{ll\_last\_size}(x,y,size) \vdash y\mapsto c_1(\texttt{null})$	1	NA	NA	0.23
31	$\texttt{zip2}(x,y) \vdash \texttt{ll}(x)$	1	NA	NA	0.23
32	$\texttt{zip2n}(x,y,size) \vdash \texttt{lln}(x,size)$	1	NA	NA	0.46
33	$\texttt{zip3}(x,y) \vdash \texttt{ll}(y)$	1	NA	NA	0.39
34	$\texttt{sortll}(x,min)*\texttt{sortll}(y,min_1) \vdash \texttt{ll}(x)$	1	NA	NA	0.13
35	$\texttt{sortlln}(x,min1,size1)* \texttt{slln}(y,min2,size2) \vdash \texttt{sll}(x,min1)$	1	NA	NA	0.27
36	$\texttt{tll}(x,ll,lr,size)*\texttt{ll}(y) \vdash \texttt{btn}(x,size)$	1	NA	NA	0.56

proofs in the combined domain. Hence, these problems are beyond the capability of $\texttt{Cyclic}_{SL}$. Ent 20 which requires mutual induction reasoning is the motivating example of `songbird` (agumented with size property) [40]. In particular, `sortll` represents a sorted list with smallest value `min`, and tll is a binary tree whose nodes point to their parents and leaves are linked by a linked list [19,24]. `S2ENT` solves each entailment incrementally: shape-based frame and then pure properties. The results show that `S2ENT` was more effective and efficient than `songbird`.

Each entailment check in Ent 29-36 requires both inductive reasoning and frame inference. These checks are beyond the capability of all existing entailment procedures for SL. `S2ENT` generates frame axioms for inductive reasoning. The experiments show that the proposed proof system can support efficient and effective reasoning on both shape and numeric domains as well as inductive proofs and frame inference.

Modular Verification for Memory Safety We enhance the existing program verifier `S2` [24] with `S2ENT` to automatically verify a range of heap-manipulating programs. We evaluate the enhanced `S2` on the C library Glib open source [16] which includes non-GUI code from the GTK+ toolkit and the GNOME desktop environment. We conduct

experiments on heap-manipulating files, i.e., singly-linked lists (gslist.c), doubly-linked lists (glist.c), balanced binary trees (gtree.c) and N-ary trees (gnode.c). These files contain fairly complex algorithms (e.g., sortedness) and the data structures used in gtree.c and gnode.c are very complex. Some procedures of gslist.c and glist.c were evaluated by tools presented in [36,31,9] where the user had to manually provide a large number of lemmas to support the tool. Furthermore, the verification in [9] is semi-automatic, i.e., verification conditions were manually generated. Besides the tool in [9], tools in [36,31] were no longer available for comparison.

Table 2. Experiments on Glib Library

	LOC	#Pr	wo.		w.		
			$\#\surd$	sec.	#syn	$\#\surd$	sec.
gslist.c	698	52	41	8.93	126	47	12.47
glist.c	784	51	39	19.41	132	46	30.01
gtree.c	1204	40	36	57.31	96	36	60.88
gnode.c	1128	65	52	37.78	174	53	53.40

In Table 2 we show, for each file the number of lines of code (excluding comments) LOC and the number of procedures #Pr. We remark that these procedures include tail-recursive procedures which are translated from loops. The columns ($\#\surd$) (and sec.) show the number of procedures (and time in seconds) for which `S2` can verify memory safety without (wo.) and with (w.) `S2ENT`. Column #syn shows the number of synthesized lemmas that used the technique in Sec. 5. With the lemma synthesis, the number of procedures that can be successfully verified increases from 168 (81%) to 182 (88%) with a time overhead of 28% (157secs/123secs).

A closer look shows that with `S2ENT` we are able to verify a number of challenging methods in gslist.c and glist.c. By generating separating lemmas, `S2ENT` successfully infers shape specifications of methods manipulating the last element of lists (i.e., `g_slist_concat` in `gslist.c` and `g_list_append` in `glist.c`). By generating equivalence lemmas, matching a newly-inferred inductive predicate with predefined predicates in `S2` is now extended beyond the shape-only domain. Moreover, the experimental results also show that the enhanced `S2` were able to verify 41/52 procedures in gslist.c and 39/51 procedures in glist.c. In comparison, while the tool in [9] could semi-automatically verify 11 procedures in gslist.c and 6 procedures in glist.c, with user-supplied lemmas the tool in [31] could verify 22 procedures in gslist.c and 10 procedures in glist.c.

7 Related Work and Conclusion

This work is related to three groups of work. The first group are those on entailment procedures in SL. Initial proof systems in SL mainly focus on a decidable fragment combining linked lists (and trees) [1,11,32,33,29,13,17,14,22,7]. Recently, Iosif *et. al.* extend the decidable fragment to restricted inductive predicates [19]. Timos et. al. [42] present a comprehensive summary on computational complexity for entailments in SL with inductive predicates. Smallfoot [1] and GRASShopper [33] provide systematic approaches for frame inference but with limited support for (general) inductive predicates. Extending these approaches to support general inductive predicates is non-trivial. GRASShopper is limited to a GRASS-reducible class of inductive predicates. While Smallfoot system has been designed to allow the use of general inductive predicates, the inference rules in Smallfoot are hardwired for list predicates only and a set of new rules

must be developed for a proof system targeting general inductive predicates. SLEEK [8] and jStar [12] support frame inference with a soundness guarantee for general inductive predicates. However, they provide limited support for induction using user-supplied lemmas [30,12]. Our work, like [8,36], targets an undecidable SL fragment including (arbitrary) inductive predicates and numerical constraints; we trade completeness for expressiveness. In addition to what are supported in [8,36], we support frame inference with inductive reasoning in SL by providing a system of cyclic proofs.

The second group is work on inductive reasoning. Lemmas are used to enhance the inductive reasoning of heap-based programs [30,5,12]. They are used as alternative unfoldings beyond predicates' definitions [30,5], external inference rules [12], or intelligent generalization to support inductive reasoning [3]. Unfortunately, the mechanisms in these systems require users to supply those additional lemmas that might be needed during a proof. SPEN [15] synthesizes lemmas to enhance inductive reasoning for some inductive predicates with bags of values. However, it is designed to support some specific classes of inductive predicates and it is difficult to extend it to cater for general inductive predicates. For a solution to inductive reasoning in SL, Smallfoot [1,3,5] presents subtraction rules that are consequent from a set of lemmas of lists and trees. Brotherston *et. al.* propose cyclic proof system for the entailment problem [2,3]. Similarly, the circularity rule has been introduced in matching logic [38], Constraint Logic Programming [9] and separation logic combined with predicate definitions and arithmetic [40]. Furthermore, work in [39] supports frame inference based on an ad-hoc mechanism, using a simple unfolding and matching. Like [3,9,40], our system also uses historical sequents at case split steps as induction hypotheses. Beyond these systems [3,9,15,40], `S2ENT` infers frames for inductive proofs systematically; and thus it gives a better support for modular verification of heap-manipulating programs. Moreover, we show how we can incrementally support inductive reasoning for the combination of heap and pure domains. In contrast, there are no formalized discussions in [5,9,40] about inductive reasoning for the combined domains; while [5] supports these domains separately, [9,40] only demonstrates their support through experimental results.

The third group is on lemma synthesis. In inductive reasoning, auxiliary lemmas are generated to discover theorems (e.g. [23,10,28]). The key elements of these techniques are heuristics used to generate equivalent lemmas for sets of given functions, constants and datatypes. In our work, we introduce lemma synthesis to strengthen the inductive constraints. To support theorem discovery, we synthesize equivalent and separating lemmas. This mechanism can be extended to other heuristics to enhance the completeness of modular verification.

Conclusion We have presented a novel approach to frame inference for inductive entailments in SL with inductive predicates and arithmetic. The core of our proposal is the system of lemma synthesis through cyclic proofs in which back-links are formed using the cut rule. Moreover, we have presented two extensions to strengthen the inferred frames. Our evaluation indicates that our system is able to infer frame axioms for inductive entailment checking that are beyond the capability of the existing systems.

Acknowledgements. This research is partially supported by project T2MOE1704 from Ministry of Education, Singapore.

References

1. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*, volume 3780, pages 52–68, November 2005.
2. J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *Proceedings of TABLEAUX-14*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.
3. J. Brotherston, D. Distefano, and R. L. Petersen. Automated cyclic entailment proofs in separation logic. In *CADE*, pages 131–146, 2011.
4. J. Brotherston, C. Fuhs, N. Gorogiannis, and J. N. Pérez. A decision procedure for satisfiability in separation logic with inductive predicates. In *Proceedings of CSL-LICS*, 2014.
5. J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *Proceedings of APLAS-10*, LNCS, pages 350–367. Springer, 2012.
6. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–300. ACM, 2009.
7. T. Chen, F. Song, and Z. Wu. Tractability of Separation Logic with Inductive Definitions: Beyond Lists. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory (CONCUR 2017)*, volume 85, pages 37:1–37:17, 2017.
8. W.N. Chin, C. David, H.H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *SCP*, 77(9):1006–1036, 2012.
9. D.-H. Chu, J. Jaffar, and M.-T. Trinh. Automatic induction proofs of data-structures in imperative programs. *PLDI'15*, 2015.
10. Koen Claessen, Moa Johansson, Dan Rosn, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *CADE '24*, volume 7898, pages 392–406. 2013.
11. B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, volume 6901, pages 235–249. 2011.
12. D. Distefano and M. Parkinson. jstar: Towards practical verification for java. In *OOPSLA '08*, pages 213–226, New York, NY, USA, 2008. ACM.
13. Constantin Enea, Ondřej Lengál, Mihaela Sighireanu, and Tomáš Vojnar. Compositional entailment checking for a fragment of separation logic. In Jacques Garrigue, editor, *APLAS 2014: Programming Languages and Systems*, pages 314–333, 2014.
14. Constantin Enea, Ondrej Lengál, Mihaela Sighireanu, and Tomás Vojnar. Compositional entailment checking for a fragment of separation logic. *Formal Methods in System Design*, 51(3):575–607, 2017.
15. Constantin Enea, Mihaela Sighireanu, and Zhilin Wu. On automated lemma generation for separation logic with inductive definitions. *ATVA*, 2015.
16. Glib. version 2.38.2. https://developer.gnome.org/glib/, 2013. [Online; accessed 13-Oct-2017].
17. X. Gu, T. Chen, and Z. Wu. A complete decision procedure for linearly compositional separation logic with data constraints. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning: 8th International Joint Conference, IJCAR 2016*, pages 532–549, 2016.
18. L. Holik, O. Lengál, A. Rogalewicz, J. Simácek, and T. Vojnar. Fully automated shape analysis based on forest automata. In *CAV'13*, pages 740–755, 2013.
19. R. Iosif, A. Rogalewicz, and J. Simácek. The tree width of separation logic with recursive definitions. In *CADE*, pages 21–38, 2013.
20. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM POPL*, pages 14–26, London, January 2001.
21. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NFM*, pages 41–55. 2011.

22. C. Jansen, J. Katelaan, C. Matheja, T. Noll, and F. Zuleger. Unified reasoning about robustness properties of symbolic-heap separation logic. In Hongseok Yang, editor, *ESOP: Programming Languages and Systems*, pages 611–638. Springer Berlin Heidelberg, 2017.
23. Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *J. Autom. Reason.*, 47(3):251–289, October 2011.
24. Q. L. Le, C. Gherghina, S. Qin, and W.-N. Chin. Shape analysis via second-order bi-abduction. In *CAV*, volume 8559, pages 52–68. 2014.
25. Q. L. Le, J. Sun, and W.-N. Chin. Satisfiability modulo heap-based programs. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 382–404. Springer International Publishing, Cham, 2016.
26. Q. L. Le, M. Tatsuta, J. Sun, and W.-N. Chin. A decidable fragment in separation logic withinductive predicates and arithmetic. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification: 29th International Conference*, pages 495–517, 2017.
27. Leonardo M. and Nikolaj B. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
28. Roy McCasland, Alan Bundy, and Autexier Serge. Automated discovery of inductive theorems. *Studies in Logic, Grammar and Rhetoric*, 10(23), 2007.
29. JuanAntonio Navarro Prez and Andrey Rybalchenko. Separation logic modulo theories. In *APLAS*, volume 8301, pages 90–106. 2013.
30. H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *CAV*, pages 355–369, 2008.
31. E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in c using separation logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 440–451, New York, NY, USA, 2014. ACM.
32. Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566, 2011.
33. R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044, pages 773–789. 2013.
34. C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345, 2006.
35. S. Qin, G. He, W.-N. Chin, F. Craciun, M. He, and Z. Ming. Automated specification inference in a combined domain via user-defined predicates. *Sci. Comput. Program.*, 148(C):189–212, November 2017.
36. X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, New York, NY, USA, 2013. ACM.
37. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.
38. Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *OOPSLA '12*, pages 555–574, New York, NY, USA, 2012. ACM.
39. Reuben N. S. Rowe and James Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. CPP, 2017.
40. Q.-T. Ta, T. C. Le, S.-C. Khoo, and W.-N. Chin. Automated mutual explicit induction proof in separation logic. In *FM*. 2016.
41. M. Tatsuta, Q. L. Le, and W.-N. Chin. Decision procedure for separation logic with inductive predicates and presburger arithmetic. In *APLAS*. 2016.
42. A. Timos, G. Nikos, H. Christoph, K. Max, and O. Joël. Foundations for decision problems in separation logic with general inductive predicates. In *FoSSaCS*, pages 411–425, 2014.
43. M.-T. Trinh, Q. L. Le, C. David, and W.-N. Chin. Bi-abduction with pure properties for specification inference. In *APLAS*, pages 107–123. 2013.
44. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.