Automated Mutual Explicit Induction Proof in Separation Logic

Quang-Trung Ta(✉), Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin

School of Computing, National University of Singapore, Singapore, Singapore
{taqt,chanhle,khoosc,chinwn}@comp.nus.edu.sg

Abstract. We present a sequent-based deductive system for automatically proving entailments in separation logic by using mathematical induction. Our technique, called *mutual explicit induction proof*, is an instance of Noetherian induction. Specifically, we propose a novel induction principle on a well-founded relation of separation logic model, and follow the *explicit* induction methods to implement this principle as inference rules, so that it can be easily integrated into a deductive system. We also support *mutual* induction, a natural feature of implicit induction, where the goal entailment and other entailments derived during the proof search can be used as hypotheses to prove each other. We have implemented a prototype prover and evaluated it on a benchmark of handcrafted entailments as well as benchmarks from a separation logic competition.

1 Introduction

Separation logic (SL) [22,30] has been actively used recently to reason about imperative programs that alter data structures. For example, the static analysis tool Infer [15] of Facebook has been using SL to discover critical memory safety bugs in Android and iOS applications. One of the pivotal features making the success of SL is the *separating conjunction* operator ($*$), which is used to describe the separation of computer memory. In particular, the assertion $p * q$ denotes a memory portion which can be decomposed into two *disjoint* sub-portions held by p and q, respectively. In addition, SL is also equipped with the ability for users to define inductive heap predicates [4,16,29]. The combination of the separating conjunction and inductive heap predicates makes SL expressive enough to model various types of recursive data structures, such as linked lists and trees.

However, this powerful expressiveness also poses challenges in reasoning about SL entailments. Considerable researches have been conducted on the SL entailment proving problem, including the works [4,7,11] related to mathematical induction. In particular, Brotherston et al. [4,7] propose the *cyclic proof*, which allows proof trees to contain cycles, and can be perceived as infinite derivation trees. Furthermore, during the proof derivation, induction hypotheses are not explicitly identified via applications of induction rules; instead, they are implicitly obtained via the discovery of valid cycle proofs. Consequently, a

J. Fitzgerald et al. (Eds.): FM 2016, LNCS 9995, pp. 659–676, 2016.
DOI: 10.1007/978-3-319-48989-6_40

soundness condition needs to be checked globally on proof trees. On the other hand, Chu et al. [11] apply *structural induction* on inductive heap predicates for proving SL entailments. During proof search, this technique dynamically uses derived entailments as induction hypotheses. When applying induction hypotheses, it performs a local check to ensure that predicates in the target entailments are substructures of predicates in the entailments captured as hypotheses. This dynamicity in hypothesis generation enables multiple induction hypotheses within a single proof path to be exploited; however, it does not admit hypotheses obtained from different proof paths.

In this work, we develop a sequent-based deductive system for proving SL entailments by using mathematical induction. Our technique is an instance of Noetherian induction [8], where we propose a novel induction principle based on a well-founded relation of SL models. Generally, proof techniques based on Noetherian induction are often classified into two categories, i.e., *explicit* and *implicit* induction [8], and each of them presents advantages over the other. We follow the explicit induction methods to implement the induction principle as inference rules, so that it can be easily integrated into a deductive system, and the soundness condition can be checked locally in each application of inference rules. In addition, since the well-founded relation defined in our induction principle does not depend directly on the substructure relationship, induction hypotheses gathered in one proof path can be used for hypothesis applications at other proof paths of the entire proof tree. Thus, our induction principle also favors *mutual induction*, a natural feature of *implicit induction*, in which the goal entailment and other entailments derived during the proof search can be used as hypotheses to prove each other. Our proof technique, therefore, does not restrict induction hypotheses to be collected from only one proof path, but rather from all derived paths of the proof tree.

Related Work. The entailment proving problem in SL has been actively studied recently. Various sound and complete techniques have been introduced, but they deal with only *pre-defined* inductive heap predicates, whose definitions and semantics are given in advance [1–3,12,23–26]. Since these techniques are designated to only certain classes of pre-defined predicates, they are not suitable for handling general inductive heap predicates.

Iosif et al. [16,17] and Enea et al. [13] aim to prove entailments in more general SL fragments by translating SL assertions into tree automata. However, these approaches still have certain restrictions on inductive heap predicates, such as the predicates must have the *bounded tree width* property, or they are variants of linked list structures. Proof techniques proposed by Nguyen et al. [10,20,21] and by Madhusudan et al. [27] can prove SL entailments with *general* inductive heap predicates. Nonetheless, these techniques are semi-automated since users are required to provide supplementing lemmas to assist in handling those predicates. In [14], Enea et al. develop a mechanism to automatically synthesize these supporting lemmas, but solely limited to certain kinds of lemmas, i.e., *composition lemmas*, *completion lemmas* and *stronger lemmas*.

Cyclic proof [4,7] and induction proof in [11] are most closely related to our approach. We recall the aforementioned comments that cyclic proof requires soundness condition to be checked globally on proof trees, whereas proof technique in [11] restricts that induction hypotheses collected from one path of proof tree cannot be used to prove entailments in other paths. Our work differs from them as we not only allow soundness condition to be checked locally at inference rule level, but also support mutual induction where entailments from different proof paths can be used as hypotheses to prove each other.

Contribution. Our contributions in this work are summarized as follows:

– We define a well-founded relation on SL models and use it to construct a novel mutual induction principle for proving SL entailments.
– We develop a deductive system for proving SL entailments based on the proposed mutual induction principle, and prove soundness of the proof system.
– We implement a prototype prover, named Songbird, and experiment on it with benchmarks of handcrafted entailments as well as entailments collected from SL-COMP, an SL competition. Our prover is available for both online use and download at: http://loris-5.d2.comp.nus.edu.sg/songbird/.

2 Motivating Example

We consider the procedure `traverse` in Fig. 1, which traverses a linked list in an unusual way, by randomly jumping either one or two steps at a time. In order to verify memory safety of this program, automated verification tools such as [5,9,18] will first formulate the shape of the computer memory manipulated by `traverse`. Suppose the initially discovered shape is represented by an *inductive heap predicate* $\mathsf{tmp}(x)$ in SL, defined as:

$$\mathsf{tmp}(x) \;\triangleq\; \mathsf{emp} \;\vee\; \exists u.(x{\mapsto}u * \mathsf{tmp}(u)) \;\vee\; \exists u,v.(x{\mapsto}u * u{\mapsto}v * \mathsf{tmp}(v))$$

Intuitively, $\mathsf{tmp}(x)$ covers three possible cases of the shape, which can be an empty memory emp (when `x == NULL`), or be recursively expanded by a single data structure $x{\mapsto}u$ (when `traverse` jumps one step), or be recursively expanded by two structures $x{\mapsto}u$ and $u{\mapsto}v$ (when `traverse` jumps two steps). Note that $x{\mapsto}u$ and $u{\mapsto}v$ are SL predicates modeling the data structure `node`. Details about the SL syntax will be explained in Sect. 3.

```
struct node {struct node *next;}
void traverse(struct node *x){
    if (x == NULL) return;
    bool jump = random();
    if (jump && x→next != NULL)
        traverse(x→next→next);
    else traverse(x→next); }
```

Fig. 1. A linked-list traversal algorithm with random jump

Since the derived shape is anomalous, the verifiers or users may want to examine if it is actually a linked list segment, modeled by the following predicate:

$$\mathsf{ls}(x,y) \;\triangleq\; (\mathsf{emp} \wedge x = y) \vee \exists w.(x{\mapsto}w * \mathsf{ls}(w,y))$$

$$\begin{array}{c l}
\hline
\mathit{true} \vdash \exists y, w.\,(u{=}w \wedge t{=}y) & (\vdash_{\text{pure}})\text{: Valid, proved by external provers, e.g. Z3.}\\
\hline
\mathsf{ls}(u,t) \vdash \exists y, w.\,(\mathsf{ls}(w,y) \wedge u{=}w) & ({*}\,\mathsf{P})\text{: Match and remove predicates } \mathsf{ls}(u,t) \text{ and } \mathsf{ls}(w,y).\\
\hline
x{\mapsto}u * \mathsf{ls}(u,t) \vdash \exists y, w.\,(x{\mapsto}w * \mathsf{ls}(w,y)) & ({*}\mapsto)\text{: Match and remove data nodes } x{\mapsto}u \text{ and } x{\mapsto}w.\\
\hline
(E_4)\ x{\mapsto}u * \mathsf{ls}(u,t) \vdash \exists y.\,\mathsf{ls}(x,y) & (\mathsf{PR})\text{: Unfold } \mathsf{ls}(x,y) \text{ by its inductive case.}\\
\hline
(E_2)\ x{\mapsto}u * \mathsf{tmp}(u) \vdash \exists y.\,\mathsf{ls}(x,y) & (\mathsf{AH})\text{: Apply IH } E \text{ with subst. } [u/x]\text{, rename } y \text{ to fresh } t.
\end{array}$$

Fig. 2. Proof tree of E_2, using induction hypothesis E

This can be done by checking the validity of the following entailment:

$$E \triangleq \mathsf{tmp}(x) \vdash \exists y.\,\mathsf{ls}(x,y)$$

In the semantics of SL, the entailment E is said to be *valid*, if all memory models satisfying $\mathsf{tmp}(x)$ also satisfy $\exists y.\,\mathsf{ls}(x,y)$. To prove it by induction, E is firstly recorded as an induction hypothesis (IH), then the predicate $\mathsf{tmp}(x)$ is analyzed in each case of its definition, via a method called unfolding, to derive new entailments E_1, E_2, E_3 as follows.

$$E_1 \triangleq \mathsf{emp} \vdash \exists y.\,\mathsf{ls}(x,y) \qquad E_2 \triangleq x{\mapsto}u * \mathsf{tmp}(u) \vdash \exists y.\,\mathsf{ls}(x,y)$$
$$E_3 \triangleq x{\mapsto}u * u{\mapsto}v * \mathsf{tmp}(v) \vdash \exists y.\,\mathsf{ls}(x,y)$$

The entailment E_1 can be easily proved by unfolding the predicate $\mathsf{ls}(x,y)$ in the right side by its base case to obtain a valid entailment $\mathsf{emp} \vdash \exists y.(\mathsf{emp} \wedge x = y)$. On the contrary, the entailment E_2 can only be proved by using the induction hypothesis E. Its (simplified) proof tree can be depicted in Fig. 2.

We can also prove E_3 by the same method, i.e., applying the IH E, and its proof tree is shown in Fig. 3.

Using a different strategy, we observe that once E_2 is proved, entailments derived during its proof, i.e., E_2 and E_4, can be used as hypotheses to prove E_3. In this case, the new proof of E_3 is much simpler than the above original

$$\begin{array}{c l}
\hline
\mathit{true} \vdash \exists y, z, w.\,(u{=}z \wedge v{=}w \wedge t{=}y) & (\vdash_{\text{pure}})\text{: Valid, proved by external prover, e.g. Z3.}\\
\hline
\mathsf{ls}(v,t) \vdash \exists y, z, w.\,(\mathsf{ls}(w,y) \wedge u{=}z \wedge v{=}w) & ({*}\,\mathsf{P})\text{: Remove predicates } \mathsf{ls}(v,t) \text{ and } \mathsf{ls}(w,y).\\
\hline
u{\mapsto}v * \mathsf{ls}(v,t) \vdash \exists y, z, w.\,(z{\mapsto}w * \mathsf{ls}(w,y) \wedge u{=}z) & ({*}\mapsto)\text{: Remove data nodes } u{\mapsto}v \text{ and } z{\mapsto}w.\\
\hline
u{\mapsto}v * \mathsf{ls}(v,t) \vdash \exists y, z.\,(\mathsf{ls}(z,y) \wedge u{=}z) & (\mathsf{PR})\text{: Unfolding } \mathsf{ls}(z,y) \text{ by inductive case.}\\
\hline
x{\mapsto}u * u{\mapsto}v * \mathsf{ls}(v,t) \vdash \exists y, z.\,(x{\mapsto}z * \mathsf{ls}(z,y)) & ({*}\mapsto)\text{: Remove data nodes } x{\mapsto}u \text{ and } x{\mapsto}z.\\
\hline
x{\mapsto}u * u{\mapsto}v * \mathsf{ls}(v,t) \vdash \exists y.\,\mathsf{ls}(x,y) & (\mathsf{PR})\text{: Unfold } \mathsf{ls}(x,y) \text{ by inductive case.}\\
\hline
(E_3)\ x{\mapsto}u * u{\mapsto}v * \mathsf{tmp}(v) \vdash \exists y.\,\mathsf{ls}(x,y) & (\mathsf{AH})\text{: Apply IH } E \text{ with substitution } [v/x]\text{, and rename } y \text{ to } t
\end{array}$$

Fig. 3. Ordinary proof tree of E_3, using induction hypothesis E

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathit{true} \vdash \exists y.\, y = z}\ (\vdash_{\text{pure}})\text{: Valid, proved by external provers, e.g., Z3.}}{\mathsf{ls}(x,z) \vdash \exists y.\, \mathsf{ls}(x,y)}\ ({*}\,\mathsf{P})\text{: Remove predicates } \mathsf{ls}(x,z) \text{ and } \mathsf{ls}(x,y).}{x{\mapsto}u * \mathsf{ls}(u,r) \vdash \exists y.\, \mathsf{ls}(x,y)}\ (\mathsf{AH})\text{: Apply } E_4 \text{ with subst. } [r/t] \text{, and rename } y \text{ to } z.}{(E_3)\ x{\mapsto}u * u{\mapsto}v * \mathsf{tmp}(v) \vdash \exists y.\, \mathsf{ls}(x,y)}\ (\mathsf{AH})\text{: Apply hypothesis } E_2 \text{ with subst. } [u/x, v/u]\text{, and rename } y \text{ to } r.$$

Fig. 4. New proof tree of E_3, using hypotheses E_2 and E_4

induction proof, as demonstrated in Fig. 4; the proving process, therefore, is more efficient.

In the new proof tree, the entailment E_4 can be directly used as a hypothesis to prove other entailments since it is already proven *valid* (see Fig. 2). However, when E_2 is applied to prove E_3, thus prove E, it is not straightforward to conclude about E, since the validity of E_2 is still *unknown*. This is because the proof of E_2 in Fig. 2 also uses E as a hypothesis. Therefore, E and E_2 jointly form a *mutual induction* proof, in which they can be used to prove each other. The theoretical principle of this proof technique will be introduced in Sect. 4.

3 Theoretical Background

In this work, we consider the *symbolic-heap* fragment of separation logic with arbitrary user-defined inductive heap predicates. We denote this logic fragment as SL_{ID}. It is similar to those introduced in [6,16], but extended with linear arithmetic (LA) to describe more expressive properties of the data structures, such as size or sortedness. The syntax and semantics of the SL_{ID} assertions and their entailments are introduced in this section.

3.1 Symbolic-Heap Separation Logic

Syntax. The syntax of our considered separation logic fragment SL_{ID} is described in Fig. 5. In particular, the predicate emp represents an *empty* memory. The *singleton* heap predicate $x \overset{\iota}{\mapsto} x_1,...,x_n$ models an n-field single data structure in memory where x points-to; its data type is represented by a unique *sort* ι[1] and values of its fields are captured by $x_1, ..., x_n$. The *inductive* heap predicate $\mathsf{P}(x_1,...,x_n)$ models a recursively defined data structure, which is formally defined in Definition 1. These three heap predicates, called *spatial atoms*, compose the *spatial* assertions Σ via the separating conjunction operator $*$. Π denotes *pure* assertions in linear arithmetic, which do not contain any spatial atoms.

Definition 1 (Inductive Heap Predicate). *A system of k inductive heap predicates P_i of arity n_i and parameters $x^i_1, ..., x^i_{n_i}$, with $i = 1, ..., k$, are syntac-*

[1] Note that for the simplicity of presenting the motivating example, we have removed the sort ι from the SL singleton heap predicate denoting the data structure `node`.

c, x, ι, P resp. denote constants, variables, data sorts, and predicate symbols.

$$\begin{array}{lcll}
e & ::= & c \mid x \mid -e \mid e_1{+}e_2 \mid e_1{-}e_2 & \text{Integer expressions} \\
a & ::= & \mathsf{nil} \mid x & \text{Spatial expressions} \\
\Pi & ::= & a_1 = a_2 \mid a_1 \neq a_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \mid & \text{Pure assertions} \\
 & & e_1 > e_2 \mid e_1 \geq e_2 \mid e_1 < e_2 \mid e_1 \leq e_2 \mid & \\
 & & \neg\Pi \mid \Pi_1 \wedge \Pi_2 \mid \Pi_1 \vee \Pi_2 \mid \Pi_1 \Rightarrow \Pi_2 \mid \forall x.\Pi \mid \exists x.\Pi & \\
\Sigma & ::= & \mathsf{emp} \mid x \overset{\iota}{\mapsto} x_1,...,x_n \mid \mathsf{P}(x_1,...,x_n) \mid \Sigma_1 * \Sigma_2 & \text{Spatial assertions} \\
F & ::= & \Sigma \mid \Pi \mid \Sigma \wedge \Pi \mid \exists x.F & \text{SL}_{\mathsf{ID}} \text{ assertions}
\end{array}$$

Fig. 5. Syntax of assertions in SL$_{\mathsf{ID}}$

tically defined as follows:

$$\left\{ \mathsf{P}_i(x^i_1,...,x^i_{n_i}) \;\;\triangleq\;\; F^i_1(x^i_1,...,x^i_{n_i}) \vee \cdots \vee F^i_{m_i}(x^i_1,...,x^i_{n_i}) \right\}_{i=1}^{k}$$

where $F^i_j(x^i_1,...,x^i_{n_i})$*, with* $1 \leq j \leq m_i$*, is a* definition case *of* $\mathsf{P}_i(x^i_1,...,x^i_{n_i})$*. Moreover,* F^i_j *is a* base case *of* P_i*, if it does not contain any predicate symbol which is (mutually) recursively defined with* P_i*; otherwise, it is an* inductive case*.*

Definition 2 (Syntactic Equivalence). *The syntactical equivalence relation of two spatial assertions* Σ_1 *and* Σ_2*, denoted as* $\Sigma_1 \cong \Sigma_2$*, is recursively defined as follows:*

$-\mathsf{emp} \cong \mathsf{emp}$ $\quad -u \overset{\iota}{\mapsto} v_1,...,v_n \cong u \overset{\iota}{\mapsto} v_1,...,v_n$ $\quad -\mathsf{P}(u_1,...,u_n) \cong \mathsf{P}(u_1,...,u_n)$
$-$*If* $\Sigma_1 \cong \Sigma'_1$ *and* $\Sigma_2 \cong \Sigma'_2$*, then* $\Sigma_1 * \Sigma_2 \cong \Sigma'_1 * \Sigma'_2$ *and* $\Sigma_1 * \Sigma_2 \cong \Sigma'_2 * \Sigma'_1$

Semantics. The semantics of SL$_{\mathsf{ID}}$ assertions are given in Fig. 6. Given a set `Var` of variables, `Sort` of sorts, `Val` of values and `Loc` $\subset$ `Val` of memory addresses, a model of an assertion consists of:

– a *stack* model s, which is a function s: `Var` $\rightarrow$ `Val`. We write $[\![\Pi]\!]_s$ to denote valuation of a pure assertion Π under the stack model s. Note that the constant $\mathsf{nil} \in$ `Val`$\backslash$`Loc` denotes dangling memory address.
– a *heap* model h, which is a partial function h: `Loc` $\rightharpoonup_{\texttt{fin}}$ (`Sort` $\rightarrow$ (`Val list`)). dom(h) denotes domain of h, and $|h|$ is cardinality of dom(h). We follow Reynolds' semantics [28] to consider *finite* heap models, i.e., $|h| < \infty$. $h \# h'$ indicates that h and h' have disjoint domains, i.e., $\text{dom}(h) \cap \text{dom}(h') = \varnothing$, and $h \circ h'$ is the union of two disjoint heap models h, h', i.e., $h \# h'$.

3.2 Entailments in SL$_{\mathsf{ID}}$

In this section, we formally define the SL$_{\mathsf{ID}}$ entailments and introduce a new concept of *model of entailments*, which will be used in the next section to construct the well-founded relation in our induction principle.

Definition 3 (Entailment). *An entailment between two assertions F and G, denoted as $F \vdash G$, is said to be* valid *(holds), iff $s, h \models F$ implies that $s, h \models G$, for all models s, h. Formally,*

$$F \vdash G \, is \, valid, \, iff \, \forall s, h.(s, h \models F \rightarrow s, \, h \models G)$$

Here, F and G are respectively called the *antecedent* and the *consequent* of the entailment. For simplicity, the entailment $F \vdash G$ can be denoted by just E, i.e., $E \triangleq F \vdash G$.

$$\begin{array}{lll}
s,h \models \Pi & \text{iff} & [\![\Pi]\!]_s = true \text{ and } \mathrm{dom}(h) = \varnothing \\
s,h \models \mathsf{emp} & \text{iff} & \mathrm{dom}(h) = \varnothing \\
s,h \models x \overset{\iota}{\mapsto} x_1,...,x_n & \text{iff} & s(x) \in \mathtt{Loc} \text{ and } \mathrm{dom}(h) = \{s(x)\} \\
 & & \qquad \text{and } h(s(x))\iota = (s(x_1), ..., s(x_n)) \\
s,h \models \mathsf{P}(x_1,...,x_n) & \text{iff} & s,h \models R_i(x_1,...,x_n), \text{ with } R_i(x_1,...,x_n) \text{ is one of} \\
 & & \qquad \text{the definition cases of } \mathsf{P}(x_1,...,x_n) \\
s,h \models \Sigma_1 * \Sigma_2 & \text{iff} & \text{there exist } h_1, h_2 \text{ such that: } h_1 \# h_2,\ h_1 \circ h_2 = h \\
 & & \qquad \text{and } s,h_1 \models \Sigma_1 \text{ and } s,h_2 \models \Sigma_2 \\
s,h \models \Sigma \wedge \Pi & \text{iff} & [\![\Pi]\!]_s = true \text{ and } s,h \models \Sigma \\
s,h \models \exists x.F & \text{iff} & \exists v \in \mathtt{Val}\,.\,[s|x{:}v], h \models F
\end{array}$$

Fig. 6. Semantics of assertions in SL$_{\mathsf{ID}}$. $[f|x{:}y]$ is a function like f except that it returns y for input x.

Definition 4 (Model and Counter-Model). *Given an entailment $E \triangleq F \vdash G$. An SL model s, h is called a* model *of E, iff $s, h \models F$ implies $s, h \models G$. On the contrary, s, h is called a* counter-model *of E, iff $s, h \models F$ and $s, h \not\models G$.*

We denote $s, h \models (F \vdash G)$, or $s, h \models E$, if s, h is a model of E. Similarly, we write $s, h \not\models (F \vdash G)$, or $s, h \not\models E$, if s, h is a counter-model of E. Given a list of n entailments $E_1, ..., E_n$, we write $s, h \models E_1, ..., E_n$ if s, h is a model of *all* $E_1, ..., E_n$, and $s, h \not\models E_1, ..., E_n$ if s, h is a counter-model of *some* $E_1, ..., E_n$.

4 Mutual Induction Proof for Separation Logic Entailment Using Model Order

In this section, we first introduce the general schema of *Noetherian induction, a.k.a. well-founded induction,* and then apply it in proving SL entailments.

Noetherian induction [8]**.** Given a conjecture $\mathcal{P}(\alpha)$, with α is a structure of type τ, the general schema of Noetherian induction on the structure α is

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

where $\prec_\tau$ is a well-founded relation on τ, i.e., there is no infinite descending chain, like $... \prec_\tau \alpha_n \prec_\tau ... \prec_\tau \alpha_2 \prec_\tau \alpha_1$. Noetherian induction can be applied for arbitrary type τ, such as data structures or control flow. However, success in proving a conjecture by induction is highly dependent on the choice of the induction variable α and the well-founded relation $\prec_\tau$.

Proving SL entailments using Noetherian induction. We observe that an SL entailment E is said to be *valid* if $s, h \models E$ for all model s, h, given that the heap domain is finite, i.e., $\forall h. |h| \in \mathbb{N}$, according to Reynolds' semantics [28]. This inspires us to define a well-founded relation among SL models, called *model order*, by comparing size of their heap domains. To prove an SL entailment by Noetherian induction based on this order, we will show that if all the smaller models satisfying the entailment implies that the bigger model also satisfies the entailment, then the entailment is satisfied by all models, thus it is valid. The model order and induction principle are formally described as follows.

Definition 5 (Model Order). *The* model order*, denoted by $\prec$, of SL models is a binary relation defined as: $s_1, h_1 \prec s_2, h_2$, if $|h_1| < |h_2|$.*

Theorem 1 (Well-Founded Relation). *The model order $\prec$ of SL models is a well-founded relation.*

Proof. By contradiction, suppose that $\prec$ were not well-founded, then there would exist an infinite descending chain: $... \prec s_n, h_n \prec ... \prec s_1, h_1$. It follows that there would exist an infinite descending chain: $... < |h_n| < ... < |h_1|$. This is impossible since domain size of heap model is finite, i.e., $|h_1|, ..., |h_n|, ... \in \mathbb{N}$. $\square$

Theorem 2 (Induction Principle). *An entailment E is valid, if for all model s, h, the following holds: $(\forall s', h'.\ s', h' \prec s, h \rightarrow s', h' \models E) \rightarrow s, h \models E$. Formally:*

$$\frac{\forall s, h.\ (\forall s', h'.\ s', h' \prec s, h \rightarrow s', h' \models E) \rightarrow s, h \models E}{\forall s, h.\ s, h \models E}$$

Since our induction principle is constructed on the SL model order, an induction hypothesis can be used in the proof of any entailment whenever the decreasing condition on model order is satisfied. This flexibility allows us to extend the aforementioned principle to support *mutual induction*, in which multiple entailments can participate in an induction proof, and each of them can be used as a hypothesis to prove the other. In the following, we will introduce our *mutual induction principle.* Note that the induction principle in Theorem 2 is an instance of this principle, when only one entailment takes part in the induction proof.

Theorem 3 (Mutual Induction Principle). *Given n entailments $E_1, ..., E_n$. All of them are valid, if for all model s, h, the following holds: $(\forall s', h'.\ s', h' \prec s, h \rightarrow s', h' \models E_1, ..., E_n) \rightarrow s, h \models E_1, ..., E_n$. Formally:*

$$\frac{\forall s,h.\ (\forall s',h'.\ s',h' \prec s,h \rightarrow s',h' \models E_1,...,E_n) \rightarrow s,h \models E_1,...,E_n}{\forall s,h.\ s,h \models E_1,...,E_n}$$

Proof. By contradiction, assume that some of $E_1,...,E_n$ were invalid. Then, there would exist some counter-models s,h such that $s,h \not\models E_1,...,E_n$. Since $\prec$ is a well-founded relation, there would exist the *least* counter-model s_1,h_1 such that $s_1,h_1 \not\models E_1,...,E_n$, and, $s_1',h_1' \models E_1,...,E_n$ for all $s_1',h_1' \prec s_1,h_1$. Following the theorem's hypothesis $\forall s,h.\ (\forall s',h'.\ s',h' \prec s,h \rightarrow s',h' \models E_1,...,E_n) \rightarrow s,h \models E_1,...,E_n$, we have $s_1,h_1 \models E_1,...,E_n$. This contradicts with the assumption that s_1,h_1 is a counter-model. □

5 The Proof System

In this section, we introduce a sequent-based deductive system, which comprises a set of inference rules depicted in Fig. 7 (logical rules) and Fig. 8 (induction rules), and a proof search procedure in Fig. 10. Each inference rule has zero or more premises, a conclusion and possibly a side condition. A premise or a conclusion is described in the same form of $\mathcal{H},\ \rho,\ F_1 \vdash F_2$, where (i) $F_1 \vdash F_2$ is an entailment, (ii) $\mathcal{H}$ is a set of entailments with validity status, which are recorded during proof search and can be used as hypotheses to prove $F_1 \vdash F_2$, and (iii) ρ is a proof trace capturing a chronological list of inference rules applied by the proof search procedure to reach $F_1 \vdash F_2$.

In addition, the entailment in the conclusion of a rule is called the *goal entailment*. Rules with zero (empty) premise is called *axiom rules*. A proof trace ρ containing n rules $R_1,\ldots,R_n$, with $n \geq 0$, is represented by $[(R_1),\ldots,(R_n)]$, where the head (R_1) of ρ is the latest rule used by the proof search procedure. In addition, some operations over proof traces are (i) insertion: $(R) :: \rho$, (ii) membership checking: $(R) \in \rho$, and (iii) concatenation: $\rho_1 \mathbin{@} \rho_2$.

5.1 Logical Rules

Logical rules in Fig. 7 deal with the logical structure of SL entailments. For brevity, in these rules, we write the *complete* symbolic-heap assertion $\exists \vec{x}.(\Sigma \wedge \Pi)$ as a *standalone* F. We define the *conjoined* assertion $F * \Sigma' \triangleq \Sigma * \Sigma' \wedge \Pi$ and $F \wedge \Pi' \triangleq \Sigma \wedge \Pi \wedge \Pi'$, given that existential quantifiers does not occur in the outermost scope of F, i.e., $F \triangleq \Sigma \wedge \Pi$. The notation $\vec{u} = \vec{v}$ means $(u_1 = v_1) \wedge \ldots \wedge (u_n = v_n)$, given that $\vec{u} = u_1,\ldots,u_n$ and $\vec{v} = v_1,\ldots,v_n$ are two lists containing the same number of variables. We also write $\vec{x} \# \vec{y}$ to denote $\vec{x}$ and $\vec{y}$ are disjoint, i.e., $\nexists u.(u \in \vec{x} \wedge u \in \vec{y})$, and use $\mathtt{FV}(F)$ to denote the list of all free variables of an assertion F. Moreover, $F[e/x]$ is a formula obtained from F by substituting the expression e for all occurrences of the free variable x in F.

The set of logical rules are explained in details as follows:

– **Axiom rules.** The rule $\vdash_{\text{pure}}$ proves a pure entailment $\Pi_1 \vdash \Pi_2$ by invoking off-the-shelf provers such as Z3 [19] to check the pure implication $\Pi_1 \Rightarrow \Pi_2$

$$(\bot\,\mathsf{L}_1)\ \frac{}{\mathcal{H},\ \rho,\ F_1 \wedge u{\neq}u \vdash F_2} \qquad (\bot\,\mathsf{L}_2)\ \frac{}{\mathcal{H},\ \rho,\ F_1 * u\overset{\iota_1}{\longmapsto}\vec{v} * u\overset{\iota_2}{\longmapsto}\vec{w} \vdash F_2}$$

$$(\vdash_{\text{pure}})\ \frac{}{\mathcal{H},\ \rho,\ \Pi_1 \vdash \Pi_2}\ \Pi_1 \Rightarrow \Pi_2 \qquad (\mathsf{empL})\ \frac{\mathcal{H},\ \rho',\ F_1 \vdash F_2}{\mathcal{H},\ \rho,\ F_1 * \mathsf{emp} \vdash F_2}$$

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

$$(=\mathsf{R})\ \frac{\mathcal{H},\ \rho',\ F_1 \vdash \exists\vec{x}.F_2}{\mathcal{H},\ \rho,\ F_1 \vdash \exists\vec{x}.(F_2 \wedge u{=}u)} \qquad (*\mapsto)\ \frac{\mathcal{H},\ \rho',\ F_1 \vdash \exists\vec{x}.(F_2 \wedge u{=}t \wedge \vec{v}{=}\ \vec{\ })\,w}{\mathcal{H},\ \rho,\ F_1 * u\overset{\iota}{\longmapsto}\vec{v} \vdash \exists\vec{x}.(F_2 * t\overset{\iota}{\longmapsto}\vec{w})}\ (u,\vec{v})\,\#\,\vec{x}$$

$$(\exists\,\mathsf{L})\ \frac{\mathcal{H},\ \rho',\ F_1[u/x] \vdash F_2}{\mathcal{H},\ \rho,\ \exists x.F_1 \vdash F_2}\ u \notin \mathtt{FV}(F_2) \qquad (*\,\mathsf{P})\ \frac{\mathcal{H},\ \rho',\ F_1 \vdash \exists\vec{x}.(F_2 \wedge \vec{u}{=}\vec{v})}{\mathcal{H},\ \rho,\ F_1 * \mathsf{P}(\vec{u}) \vdash \exists\vec{x}.(F_2 * \mathsf{P}(\vec{v}))}\ \vec{u}\,\#\,\vec{x}$$

$$(\exists\,\mathsf{R})\ \frac{\mathcal{H},\ \rho',\ F_1 \vdash F_2[e/x]}{\mathcal{H},\ \rho,\ F_1 \vdash \exists x.F_2} \qquad (\mathsf{PR})\ \frac{\mathcal{H},\ \rho',\ F_1 \vdash \exists\vec{x}.(F_2 * F_i^{\mathsf{P}}(\vec{u}))}{\mathcal{H},\ \rho,\ F_1 \vdash \exists\vec{x}.(F_2 * \mathsf{P}(\vec{u}))}\ \begin{array}{l}F_i^{\mathsf{P}}(\vec{u}) \text{ is one of the} \\ \text{definition cases of } \mathsf{P}(\vec{u})\end{array}$$

Fig. 7. Logical rules. Note that for a rule R with trace ρ in its conclusion, the trace in its premise is $\rho' \triangleq (R) :: \rho$.

in its side condition. The two rules $\bot\,\mathsf{L}_1$ and $\bot\,\mathsf{L}_2$ decide an entailment *vacuously* valid if its antecedent is unsatisfiable, i.e., the antecedent contains a contradiction ($u \neq u$) or overlaid data nodes ($u\overset{\iota_1}{\longmapsto}\vec{v} * u\overset{\iota_2}{\longmapsto}\vec{w}$).

- **Normalization rules.** These rules simplify their goal entailments by either eliminating existentially quantified variables ($\exists\,\mathsf{L}, \exists\,\mathsf{R}$), or removing equalities ($=\mathsf{L}, =\mathsf{R}$) or empty heap predicates ($\mathsf{empL}, \mathsf{empR}$) from antecedents (left side) or consequents (right side) of the entailments.
- **Frame rules.** The two rules $*\mapsto$ and $*\,\mathsf{P}$ applies the *frame property* of SL [28] to remove *identical* spatial atoms from two sides of entailments. Note that the identical condition is guaranteed by adding equality constraints of these spatial atoms' arguments into consequents of the derived entailments.
- **Unfolding rules.** The rule PR derives a new entailment by unfolding a heap predicate in the goal entailment's consequent by its inductive definition. Note that unfolding a heap predicate in the entailment's antecedent will be performed by the induction rule Ind, as discussed in the next section.

5.2 Induction Rules

Figure 8 presents inference rules implementing our mutual induction principle. The *induction* rule Ind firstly records its goal entailment as an induction hypothesis H, and unfolds an inductive heap predicate in the antecedent of H to derive new entailments. When H is inserted into the hypothesis vault $\mathcal{H}$, its status is initially assigned to **?** (*unknown*), indicating that its validity is not known at the moment. Later, the status of H will be updated to ✓ (*valid*) once the proof

$$(\mathsf{Ind})\ \frac{\mathcal{H} \cup \{(H, \mathbf{?})\},\ \rho',\ F_1 * F_1^{\mathsf{P}}(\vec{u}) \vdash F_2 \quad \ldots \quad \mathcal{H} \cup \{(H, \mathbf{?})\},\ \rho',\ F_1 * F_m^{\mathsf{P}}(\vec{u}) \vdash F_2}{\mathcal{H},\ \rho,\ F_1 * \mathsf{P}(\vec{u}) \vdash F_2}\ \dagger_{(\mathsf{Ind})}$$

Given $H \triangleq F_1 * \mathsf{P}(\vec{u}) \vdash F_2$, $\rho' = (\mathsf{Ind}) :: \rho$, and $\dagger_{(\mathsf{Ind})}$: $\mathsf{P}(\vec{u}) \triangleq F_1^{\mathsf{P}}(\vec{u}) \vee \ldots \vee F_m^{\mathsf{P}}(\vec{u})$

$$(\mathsf{AH})\ \frac{\mathcal{H} \cup \{(H, \mathit{status})\},\ (\mathsf{AH}) :: \rho,\ F_4\theta * \Sigma' \wedge \Pi_1 \vdash F_2}{\mathcal{H} \cup \{(H \triangleq \Sigma_3 \wedge \Pi_3 \vdash F_4, \mathit{status})\},\ \rho,\ \Sigma_1 \wedge \Pi_1 \vdash F_2}\ \begin{array}{l}\exists\theta, \Sigma'.(\Sigma_1 \cong \Sigma_3\theta * \Sigma' \wedge \Pi_1 \Rightarrow \Pi_3\theta),\\ \dagger_{(\mathsf{AH})}\end{array}$$

with $\dagger_{(\mathsf{AH})}$: $(\mathit{status}{=}\checkmark) \vee \exists\iota, u, \vec{v}, \Sigma''.(\Sigma' \cong u \overset{\iota}{\mapsto} \vec{v} * \Sigma'')$
$\vee\ \exists\rho_1, \rho_2.(\rho = \rho_1 @ [(* \mapsto)] @ \rho_2 \wedge (\mathsf{Ind}) \notin \rho_1 \wedge (\mathsf{Ind}) \in \rho_2)$.

Fig. 8. Induction rules

search procedure is able to prove it valid. Generally, given an entailment E and its proof tree $\mathcal{T}$, the proof search procedure concludes that E is valid if (i) every leaf of $\mathcal{T}$ is empty via applications of axiom rules, and (ii) all hypotheses used by the *apply hypothesis* rule AH must be derived in $\mathcal{T}$.

Rule AH is the key rule of our mutual induction principle, which applies an appropriate hypothesis $H \triangleq \Sigma_3 \wedge \Pi_3 \vdash F_4$ in proving its goal entailment $E \triangleq \Sigma_1 \wedge \Pi_1 \vdash F_2$. The rule firstly unifies the antecedents of H and E by a substitution θ, i.e., there exists a spatial assertion Σ' such that $\Sigma_1 \cong \Sigma_3\theta * \Sigma'$ and $\Pi_1 \Rightarrow \Pi_3\theta$. If such θ and Σ' exist, we can weaken the antecedent of E as follows $(\Sigma_1 \wedge \Pi_1) \vdash (\Sigma_3\theta * \Sigma' \wedge \Pi_3\theta \wedge \Pi_1) \vdash (F_4\theta * \Sigma' \wedge \Pi_1)$. Note that we use Reynolds's substitution law [28] to obtain $\Sigma_3\theta \wedge \Pi_3\theta \vdash F_4\theta$ from the hypothesis H. The proof system then derives the next goal entailment $F_4\theta * \Sigma' \wedge \Pi_1 \vdash F_2$ as shown in the premise of rule AH.

The side condition $\dagger_{(\mathsf{AH})}$ of rule AH ensures the decreasing condition of the mutual induction principle. In particular, suppose that the proof search procedure applies a hypothesis H in $\mathcal{H}$ to prove an entailment E via rule AH. If the status of H is $\checkmark$, denoted by the first condition in $\dagger_{(\mathsf{AH})}$, then H is already proved to be valid; thus it can be freely used to prove other entailments. Otherwise, the status of H is $\mathbf{?}$, and H may participate in a (mutual) induction proof with an entailment I in the proof path of E, as depicted in Fig. 9. Note that the entailment I has been recorded earlier as an induction hypothesis by an application of the induction rule Ind.

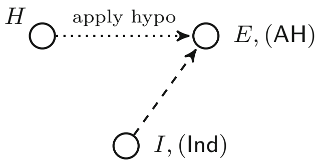

Fig. 9. Applying hypothesis

In the latter case, the induction principle requires the decrease of model size when applying the hypothesis H to prove entailment I. We then show that this decreasing condition holds if one of the following conditions of $\dagger_{(\mathsf{AH})}$ is satisfied.

(i) $\exists\iota, u, \vec{v}, \Sigma''.(\Sigma' \cong u \overset{\iota}{\mapsto} \vec{v} * \Sigma'')$ indicates that the left-over heap part Σ' after unifying antecedent of H into that of E contains at least one singleton heap predicate, or

(ii) $\exists\rho_1, \rho_2.(\rho = \rho_1@[(*\!\mapsto)]@\rho_2 \wedge (\mathsf{Ind})\notin\rho_1 \wedge (\mathsf{Ind})\in\rho_2)$ requires that there is a removal step of a singleton heap predicate by the rule $*\!\mapsto$ applied between this hypothesis application AH and the most recent induction step Ind.

Consider an arbitrary model s, h satisfying I. During the derivation path from I to E, the model s, h is transformed into a corresponding model s_e, h_e of E. We always have $|h_e| \leq |h|$ as the applications of logical rules and rule Ind never increase heap model size of entailments. Moreover, when applying H to prove E, the model s', h' of H, which corresponds to s_e, h_e of E, satisfies $|h'| \leq |h_e|$, due to the unification step in rule AH. We consider two following cases. If condition (i) is satisfied, then heap model size of the left-over part Σ' is at least 1 since Σ' contains a singleton heap predicate. As a result, $|h'| < |h_e|$ and it follows that $|h'| < |h|$. If condition (ii) is satisfied, then $|h_e| < |h|$ since there is a singleton heap predicate, whose size of heap model is 1, is removed when deriving I to E. This implies that $|h'| < |h|$. In summary, we obtain that $|h'| < |h|$ for both cases; thus, $s', h' \prec s, h$. This concludes our explanation about the rule AH.

Procedure $\mathsf{Prove}(\mathcal{H}, \rho, F \vdash G)$

Input: $\mathcal{H}, F \vdash G$ and ρ are respectively a set of hypotheses, a goal entailment and its corresponding proof trace.

Output: Validity result (True or False), a set of derived entailments with their validity statuses, and a set of hypotheses used in proof of $F \vdash G$.

```
1: $\mathcal{S} \leftarrow \{ R_{inst} \mid R_{inst} = \mathsf{Unify}(R, (\mathcal{H}, \rho, F \vdash G)) \wedge R \in \mathcal{R} \}$
2: if $\mathcal{S} = \varnothing$ then return False, $\varnothing, \varnothing$          // no rule is selected
3: for each $R_{inst}$ in $\mathcal{S}$ do
4:     if $\mathsf{GetName}(R_{inst}) \in \{\vdash_{\text{pure}}, \bot\mathsf{L}_1, \bot\mathsf{L}_2\}$ then          // R is an axiom rule
5:         return True, $\varnothing, \varnothing$
6:     $\mathcal{H}_{used} \leftarrow \varnothing$
7:     if $R_{inst}$ = AH with hypothesis $E$ then $\mathcal{H}_{used} \leftarrow \mathcal{H}_{used} \cup \{E\}$
8:     $\mathcal{H}_{derived} \leftarrow \varnothing$
9:     $(\mathcal{H}_i, \rho_i, F_i \vdash G_i)_{i=1,\ldots,n} \leftarrow \mathsf{GetPremises}(R_{inst})$          // all premises of $R_{inst}$
10:    for i = 1 to n do
11:        $res, \mathcal{H}_{derived}, \mathcal{H}'_{used} \leftarrow \mathsf{Prove}(\mathcal{H}_i \oplus \mathcal{H}_{derived}, \rho_i, F_i \vdash G_i)$
12:        if $res$ = False then return False, $\varnothing, \varnothing$
13:        $\mathcal{H}_{used} \leftarrow \mathcal{H}_{used} \cup \mathcal{H}'_{used}$
14:    if $\mathcal{H}_{used} \subseteq (\mathsf{GetEntailments}(\mathcal{H}_{derived}) \cup \{F \vdash G\})$ then
15:        $\mathcal{H}_{derived} \leftarrow \mathcal{H}_{derived} \oplus \{(F \vdash G, \checkmark)\}$
16:    else $\mathcal{H}_{derived} \leftarrow \mathcal{H}_{derived} \oplus \{(F \vdash G, ?)\}$
17:    return True, $\mathcal{H}_{derived}, \mathcal{H}_{used}$          // all derived premises are proved
18: return False, $\varnothing, \varnothing$          // all rules fail to prove $F \vdash G$
```

Fig. 10. General proof search procedure, in which $\mathcal{R}$ is the set of inference rules given in Figs. 7 and 8.

5.3 Proof Search Procedure

Our proof search procedure Prove is designed in a self-recursive manner, as presented in Fig. 10. Its inputs consist of a set of hypotheses, a proof trace, and an entailment, which are components of an inference rule's conclusion. To prove a candidate entailment $F \vdash G$, initially the hypothesis set $\mathcal{H}$ and the proof trace are assigned to empty ($\varnothing$ and $[\,]$).

Firstly, the procedure Prove finds a set $\mathcal{S}$ of suitable rules, whose conclusion can be unified with the goal entailment $F \vdash G$, among all inference rules in $\mathcal{R}$ (line 1). If no suitable rule is found, the procedure immediately returns False, indicating that it is unable to prove the entailment (line 2). Otherwise, it subsequently processes each discovered rule R_{inst} in $\mathcal{S}$ by either (i) returning True to announce a valid result, if an axiom rule is selected (line 5), or (ii) recursively searching for proofs of the derived entailments in the premises of R_{inst} (lines 9–17). In the latter case, the procedure returns False if one of the derived entailments is not proved (line 12), or returns True if all of them are proved (line 17). Finally, it simply returns False when it cannot prove the goal entailment with all selected rules (line 18).

The procedure uses a local variable $\mathcal{H}_{used}$ to store all hypotheses used during the proof search. $\mathcal{H}_{used}$ is updated when the rule AH is applied (line 7) or after the procedure finishes proving a derived entailment (lines 11 and 13). We also use another variable $\mathcal{H}_{derived}$ to capture all generated entailments with their validity statuses. The condition at line 14 checks if all hypotheses used to prove the entailment $F \vdash G$ are only introduced during the entailment's proof. If this condition is satisfied, then $F \vdash G$ is updated with a *valid status* $\checkmark$ (line 15). Otherwise, the entailment may participate in a (mutual) induction proof, thus its status is assigned to *unknown* **?** (line 16).

At line 11, the procedure uses not only the hypothesis set $\mathcal{H}_i$, introduced by the selected inference rule, but also the set $\mathcal{H}_{derived}$ containing entailments derived during proof search to prove a new goal entailment $F_i \vdash G_i$. This reflects our mutual induction principle which allows derived entailments to be used as hypotheses in other entailments' proofs. Note that the *union and update* operator $\oplus$ used in the algorithm will insert new entailments and their statuses into the set of hypotheses, or update the existing entailments with their new statuses. In addition, the auxiliary procedures used in our proof search procedure are named in a self-explanatory manner. In particular, Unify, GetName and GetPremises respectively unifies an inference rule with a goal entailment, or returns name and premises of an inference rule. Finally, GetEntailments returns all entailments stored in the set of derived entailments $\mathcal{H}_{derived}$.

Soundness. Soundness of our proof system is stated in Theorem 4. Due to page constraint, we present the detailed proof in the technical report [32].

Theorem 4 (Soundness). *Given an entailment E, if the proof search procedure returns* True *when proving E, then E is valid.*

6 Experiment

We have implemented the proposed induction proof technique into a prototype prover, named Songbird. The proof system and this paper's artifact are available for both online use and download at http://loris-5.d2.comp.nus.edu.sg/songbird/.

Category	Slide	Spen	Sleek	Cyclist	Songbird
singly-ll (64)	12	3	48	**63**	**63**
doubly-ll (37)	14	0	17	24	**26**
nested-ll (11)	0	**11**	5	6	**11**
skip-list (13)	0	**12**	4	5	7
tree (26)	12	1	14	18	**22**
Total (151)	38	27	88	116	**129**

(a)

	Songbird			
	✓$_{\text{sb}}$ ✗$_{\text{o}}$	✗$_{\text{sb}}$ ✓$_{\text{o}}$	✓$_{\text{sb}}$ ✓$_{\text{o}}$	✗$_{\text{sb}}$ ✗$_{\text{o}}$
Cyclist	13	0	116	22
Sleek	41	0	88	22
Spen	109	7	20	15
Slide	103	12	26	10

(b)

Fig. 11. Overall evaluation on the benchmark slrd_entl of SL-COMP

To evaluate our technique, we compared our system against state-of-the-art SL provers, including Slide [16,17], Spen [13], Sleek [10] and Cyclist [4,7], which had participated in the recent SL competition SL-COMP [31]. We are however unable to make direct comparison with the induction-based proof technique presented in [11] as their prover was not publicly available. Our evaluation was performed on an Ubuntu 14.04 machine with CPU Intel E5-2620 (2.4 GHz) and RAM 64 GB.

Firstly, we conduct the experiment on a set of *valid* entailments[2], collected from the benchmark slrd_entl[3] of SL-COMP. These entailments contain *general* inductive heap predicates denoting various data structures, such as singly linked lists (singly-ll), doubly linked lists (doubly-ll), nested lists (nested-ll), skip lists (skip-list) and trees (tree). We then categorize problems in this benchmark based on their predicate types. In Fig. 11(a), we report the number of entailments successfully proved by a prover in each category, with a timeout of 30 s for proving an entailment. For each category, the total number of problems is put in parentheses, and the maximum number of entailments that can be proved by the list of provers are highlighted in bold. As can be seen, Songbird can prove more entailments than all the other tools. In particular, we are the best in almost categories, except for skip-list. However, in this category, we are behind only Spen, which has been specialized for skip lists [13]. Our technique might require more effective generalization to handle the unproven skip-list examples.

In Fig. 11(b), we make a detailed comparison among Songbird and other provers. Specifically, the first column (✓$_{\text{sb}}$ ✗$_{\text{o}}$) shows the number of entailments

[2] We exclude the set of invalid entailments because some evaluated proof techniques, such as [4,10], aim to only prove validity of entailments.

[3] Available at https://github.com/mihasighi/smtcomp14-sl/tree/master/bench.

that Songbird can prove valid whereas the others cannot. The second column ($\text{✗}_{\text{sb}}\text{✓}_{\text{o}}$) reports the number of entailments that can be proved by other tools, but not by Songbird. The last two columns list the number of entailments that both Songbird and others can ($\text{✓}_{\text{sb}}\text{✓}_{\text{o}}$) or cannot ($\text{✗}_{\text{sb}}, \text{✗}_{\text{o}}$) prove. We would like to highlight that our prover efficiently proves *all* entailments proved by Cyclist (resp. Sleek) in *approximately half the time*, i.e., 20.92 vs 46.40 s for 116 entailments, in comparison with Cyclist (resp. 8.38 vs 15.50 s for 88 entailments, in comparison with Sleek). In addition, there are 13 (resp. 41) entailments that can be proved by our tool, but *not* by Cyclist (resp. Sleek). Furthermore, our Songbird outperforms Spen and Slide by more than 65 % of the total entailments, thanks to the proposed mutual induction proof technique.

Secondly, we would like to highlight the efficiency of *mutual induction* in our proof technique via a comparison between Songbird and its variant $\text{Songbird}_{\text{SI}}$, which exploits only induction hypotheses found within a *single* proof path. This mimics the structural induction technique which explores induction hypotheses in the same proof path. For this purpose, we designed a new entailment benchmark, namely slrd_ind, whose problems are more complex than those in the slrd_entl benchmark. For example, our handcrafted benchmark[4] contains an entailment $\mathsf{lsEven}(x,y) * y \mapsto z * \mathsf{lsEven}(z,t) \vdash \exists u.\, \mathsf{lsEven}(x,u) * u \mapsto t$ with the predicate $\mathsf{lsEven}(x,y)$ denoting list segments with even length. This entailment was inspired by the entailment $\mathsf{lsEven}(x,y) * \mathsf{lsEven}(y,z) \vdash \mathsf{lsEven}(x,z)$ in the problem 11.tst.smt2 of slrd_entl, contributed by team Cyclist. Note that entailments in our benchmark were constructed on the same set of linked list predicates provided in slrd_entl, comprised of regular singly linked lists (ll), linked lists with even or odd length (ll-even/odd) and linked list segments which are left- or right-recursively defined (ll-left/right). We also use a new ll2 list segment predicate whose structure is similar to the predicate tmp in our motivating example. In addition, problems in the misc. category involve all aforementioned linked list predicates.

As shown in Fig. 12, $\text{Songbird}_{\text{SI}}$ is able to prove nearly 70 % of the total entailments, which is slightly better than Cyclist[5], whereas Songbird, with full capability of mutual induction, can prove the *whole* set of entailments. This result

Category		Cyclist	$\text{Songbird}_{\text{SI}}$	Songbird
ll/ll2	(24)	18	22	24
ll-even/odd	(20)	8	17	20
ll-left/right	(20)	12	10	20
misc.	(32)	17	16	32
Total	(96)	55	65	96

Fig. 12. Comparison on slrd_ind benchmark

[4] The full benchmark is available at http://loris-5.d2.comp.nus.edu.sg/songbird/.

[5] We do not list other provers in Fig. 12 as they cannot prove any problems in slrd_ind.

is encouraging as it shows the usefulness and essentials of our mutual explicit induction proof technique in proving SL entailments.

7 Conclusion

We have proposed a novel induction technique and developed a proof system for automatically proving entailments in a fragment of SL with general inductive predicates. In essence, we show that induction can be performed on the size of the heap models of SL entailments. The implication is that, during automatic proof construction, the goal entailment and entailments derived in the entire proof tree can be used as hypotheses to prove other derived entailments, and vice versa. This novel proposal has opened up the feasibility of mutual induction in automatic proof, leading to shorter proof trees being built. In future, we would like to develop a verification system on top of the prover Songbird, so that our *mutual explicit induction* technique can be effectively used for automated verification of memory safety in imperative programs.

Acknowledgement. We would like to thank the anonymous reviewers for their valuable and helpful feedback. The first author would like to thank Dr. James Brotherston for the useful discussion about the cyclic proof. This work has been supported by NUS Research Grant R-252-000-553-112. Ton Chanh and Wei-Ngan are partially supported by MoE Tier-2 grant MOE2013-T2-2-146.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30538-5_9
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005). doi:10.1007/11575467_5
3. Bozga, M., Iosif, R., Perarnau, S.: Quantitative separation logic and programs with lists. J. Autom. Reason. **45**(2), 131–156 (2010)
4. Brotherston, J., Distefano, D., Petersen, R.L.: Automated cyclic entailment proofs in separation logic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 131–146. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22438-6_12
5. Brotherston, J., Gorogiannis, N.: Cyclic abduction of inductively defined safety and termination preconditions. In: Müller-Olm, M., Seidl, H. (eds.) SAS 2014. LNCS, vol. 8723, pp. 68–84. Springer, Heidelberg (2014). doi:10.1007/978-3-319-10936-7_5
6. Brotherston, J., Gorogiannis, N., Kanovich, M.I., Rowe, R.: Model checking for symbolic-heap separation logic with inductive predicates. In: Symposium on Principles of Programming Languages (POPL), pp. 84–96 (2016)
7. Brotherston, J., Gorogiannis, N., Petersen, R.L.: A generic cyclic theorem prover. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 350–367. Springer, Heidelberg (2012). doi:10.1007/978-3-642-35182-2_25

8. Bundy, A.: The automation of proof by mathematical induction. In: Handbook of Automated Reasoning, vol. 2, pp. 845–911 (2001)
9. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Symposium on Principles of Programming Languages (POPL), pp. 289–300 (2009)
10. Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. (SCP) **77**(9), 1006–1036 (2012)
11. Chu, D., Jaffar, J., Trinh, M.: Automatic induction proofs of data-structures in imperative programs. In: Conference on Programming Language Design and Implementation (PLDI), pp. 457–466 (2015)
12. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 235–249. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23217-6_16
13. Enea, C., Lengál, O., Sighireanu, M., Vojnar, T.: Compositional entailment checking for a fragment of separation logic. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 314–333. Springer, Heidelberg (2014). doi:10.1007/978-3-319-12736-1_17
14. Enea, C., Sighireanu, M., Wu, Z.: On automated lemma generation for separation logic with inductive definitions. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 80–96. Springer, Heidelberg (2015). doi:10.1007/978-3-319-24953-7_7
15. Infer: A tool to detect bugs in Android and iOS apps before they ship. http://fbinfer.com/. Accessed 27 May 2016
16. Iosif, R., Rogalewicz, A., Simacek, J.: The tree width of separation logic with recursive definitions. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 21–38. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38574-2_2
17. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding entailments in inductive separation logic with tree automata. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 201–218. Springer, Heidelberg (2014). doi:10.1007/978-3-319-11936-6_15
18. Le, Q.L., Gherghina, C., Qin, S., Chin, W.-N.: Shape analysis via second-order bi-abduction. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 52–68. Springer, Heidelberg (2014). doi:10.1007/978-3-319-08867-9_4
19. Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
20. Nguyen, H.H., Chin, W.-N.: Enhancing program verification with lemmas. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 355–369. Springer, Heidelberg (2008). doi:10.1007/978-3-540-70545-1_34
21. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007). doi:10.1007/978-3-540-69738-1_18
22. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). doi:10.1007/3-540-44802-0_1
23. Pérez, J.A.N., Rybalchenko, A.: Separation logic + superposition calculus = heap theorem prover. In: Conference on Programming Language Design and Implementation (PLDI), pp. 556–566 (2011)

24. Navarro Pérez, J.A., Rybalchenko, A.: Separation logic modulo theories. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 90–106. Springer, Heidelberg (2013). doi:10.1007/978-3-319-03542-0_7
25. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 773–789. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_54
26. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 711–728. Springer, Heidelberg (2014). doi:10.1007/978-3-319-08867-9_47
27. Qiu, X., Garg, P., Stefanescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: Conference on Programming Language Design and Implementation (PLDI), pp. 231–242 (2013)
28. Reynolds, J.C.: An introduction to separation logic - Lecture Notes for the PhD Fall School on Logics and Semantics of State, Copenhagen (2008). http://www.cs.cmu.edu/jcr/copenhagen08.pdf. Accessed 20 Jan 2016
29. Reynolds, J.C.: Intuitionistic reasoning about shared mutable data structure. In: Millennial Perspectives in Computer Science, Palgrave, pp. 303–321 (2000)
30. Reynolds, J.C.: Separation Logic: A logic for shared mutable data structures. In: Symposium on Logic in Computer Science (LICS), pp. 55–74 (2002)
31. Sighireanu, M., Cok, D.R.: Report on SL-COMP 2014. J. Satisf. Boolean Model. Comput. **9**, 173–186 (2016)
32. Ta, Q.T., Le, T.C., Khoo, S.C., Chin, W.N.: Automated mutual explicit induction proof in separation logic. arXiv:1609.00919 (2016)