https://doi.org/10.1007/s00165-018-0471-5
BCS © 2018
Formal Aspects of Computing (2019) 31: 207–230

Formal Aspects of Computing

Automated mutual induction proof in separation logic

Quang-Trung Ta[1], Ton Chanh Le[2], Siau-Cheng Khoo[1], Wei-Ngan Chin[1]

[1] Department of Computer Science, School of Computing, National University of Singapore, Singapore, Singapore
[2] Department of Computer Science, Stevens Institute of Technology, Hoboken, USA

Abstract. We present a deductive proof system to automatically prove separation logic entailments by mathematical induction. Our technique is called the *mutual induction proof*. It is an instance of the well-founded induction, a.k.a., Noetherian induction. More specifically, we propose a novel induction principle based on a well-founded relation of separation logic models. We implement this principle explicitly as inference rules so that it can be easily integrated into a deductive proof system. Our induction principle allows a goal entailment and other entailments derived during the proof search to be used as hypotheses to mutually prove each other. This feature increases the success chance of proving the goal entailment. We have implemented this mutual induction proof technique in a prototype prover and evaluated it on two entailment benchmarks collected from the literature as well as a synthetic benchmark. The experimental results are promising since our prover can prove most of the valid entailments in these benchmarks, and achieves a better performance than other state-of-the-art separation logic provers.

Keywords: Separation logic; Entailment proving; Mathematical induction; Mutual induction

1. Introduction

Separation logic has been actively developed in the past two decades as a promising formalism to reason about the memory-safety of imperative programs that manipulate data structures. For example, it has been utilized by industrial static analysis tools such as SLAyer [BCI11] and Infer [Cal+15] to find memory bugs in kernels and drivers of modern operating systems, system code libraries, and mobile applications. One of the pivotal features that makes the success of separation logic is the *separating conjunction* operator ($*$), which specifies the split of program memory in disjoint regions. In particular, the formula $p * q$ denotes a memory portion which can be decomposed into two *disjoint* sub-portions respectively held by p and q [Rey02]. Moreover, separation logic also allows users to define *inductive heap predicates*. The combination of the separating conjunction and inductive heap predicates enables separation logic to expressively model recursive data structures, such as linked lists, trees, and graphs [Rey02].

Correspondence and offprint requests to: Q.-T. Ta, Email: taqt@comp.nus.edu.sg

However, the above expressiveness of separation logic also poses challenges in proving the validity of entailments. These entailments are generated as verification conditions when reasoning about computer programs. Moreover, proving their validity is usually considered as the main operation in program verification. Considerable researches have been conducted on the problem of proving separation logic entailments. They include the works that are based on or similar to mathematical induction [Bro07, BDP11, CJT15]. Particularly, Brotherston et al. [Bro07, BDP11] propose a proof technique named *cyclic proof*. This technique allows a proof tree to contain cycles, which can be perceived as an infinite derivation tree. 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 cycles in the proof tree. Consequently, a soundness condition needs to be checked globally on the entire proof tree. On the other hand, Chu et al. [CJT15] present an entailment proof system based on *structural induction*. Their system dynamically utilizes entailments derived during a proof search as induction hypotheses to prove new entailments generated in the same proof path. When applying an induction hypothesis, it performs a local check to ensure that predicates in target entailments are substructures of a corresponding predicate in the induction hypothesis. However, this technique does not admit induction hypotheses obtained from different proof paths.

In this work, we develop a deductive system to prove separation logic entailments by mathematical induction. Our technique is an instance of the well-founded induction, a.k.a., Noetherian induction [Bun01], where we propose a novel induction principle based on a well-founded relation of separation logic models. Generally, proof techniques based on Noetherian induction are often classified into two categories, i.e., *explicit* and *implicit* induction, and each of them presents advantages over the other [Bun01]. 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. Also, the soundness condition can be checked locally in each inference rule application, using a selected global view of the proof tree which is collected during the proof search. Moreover, since the well-founded relation defined in our induction principle does not depend directly on any substructure relation, induction hypotheses gathered in one proof path can be applied at other paths of the proof tree. Thus, this 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 a proof tree.

Contributions. In summary, our work makes the following contributions:

- We define a well-founded relation on separation logic models and use it to construct a novel mutual induction principle to prove separation logic entailments.
- We develop a deductive system for proving entailments based on the proposed mutual induction principle. Our system can prove entailments more effectively since it allows entailments derived in a proof tree to be used as hypotheses to prove other entailments derived in the same proof.
- We implement a prototype prover, named Songbird, and experiment on it with two entailment benchmarks from a separation logic competition SL-COMP 2014 as well as a synthetic benchmark. Our experiment shows a promising result because Songbird can prove most of the entailments in these benchmarks. The prover Songbird is available online at https://songbird-prover.github.io/mutual-induction/.

Outline. The rest of our paper is organized as follows. We present a motivating example in Sect. 2 and theoretical background in Sect. 3. Afterward, we describe our main contributions, which include the mutual induction principle in Sect. 4, the formal proof system in Sect. 5, the implementation of our prototype prover in Sect. 6, and our experiment on entailment benchmarks in Sect. 7. Finally, we discuss related work in Sect. 8, limitations in Sect. 9, and conclude in Sect. 10.

```
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

Prior publication. This paper is an extended version of our earlier work published in the 21st International Conference on Formal Methods (FM) in 2016 [Ta+16]. In particular, we extend it with more explanation on theoretical background and related work in Sects. 3 and 8. We also describe more details on the implementation of our prover Songbird, by presenting two auxiliary procedures to find inference rules in Sect. 6. We discuss limitations of the proof system in Sect. 9. Finally, we re-conduct the experiments with not only the benchmark slrd_entl as in the prior publication but also an additional entailment benchmark sll_entl from SL-COMP 2014 and an improved version of our proposed benchmark slrd_ind [Ta+16]. We present these new experiments in Sect. 7.

2. Motivating example

We consider the procedure `traverse` in Fig. 1, which is written in a C-like language to traverse a linked list data structure whose elements are of type `node`. More specifically, this procedure visits the linked list's elements by randomly jumping either one or two steps at a time. In order to verify the memory safety of this procedure, an automated verification tool first needs to formulate the shape of the memory manipulated by `traverse`. Suppose the initially discovered shape is represented by an *inductive heap predicate* $\mathsf{tmp}(x)$ which is defined as follows:

$$\mathsf{tmp}(x) \quad \overset{\text{def}}{=} \quad \mathsf{emp} \quad \vee \quad \exists\, u.(x{\mapsto}u * \mathsf{tmp}(u)) \quad \vee \quad \exists\, u, v.(x{\mapsto}u * u{\mapsto}v * \mathsf{tmp}(v))$$

Intuitively, $\mathsf{tmp}(x)$ covers three possible cases of the shape, which can be an empty memory emp (when `x == NULL`), or can be recursively expanded by a single data structure $x{\mapsto}u$ (when `traverse` jumps one step), or can be recursively expanded by two structures $x{\mapsto}u$ and $u{\mapsto}v$ (when `traverse` jumps two steps). In this definition, $x{\mapsto}u$ and $u{\mapsto}v$ are two separation logic predicates modeling the data structure `node`. We will present details about these predicates and theoretical background of separation logic in Sect. 3.

Since the derived shape $\mathsf{tmp}(x)$ is anomalous, the automated verification tool may want to examine if this shape is actually a linked list segment, modeled by the following predicate:

$$\mathsf{ls}(x, y) \quad \overset{\text{def}}{=} \quad (\mathsf{emp} \wedge x{=}y) \quad \vee \quad \exists\, w.(x{\mapsto}w * \mathsf{ls}(w, y))$$

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

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

In the semantics of separation logic, the entailment E is said to be *valid*, if all memory models that satisfy $\mathsf{tmp}(x)$ also satisfy $\exists\, y.\, \mathsf{ls}(x, y)$. Since this entailment contains the inductive heap predicate $\mathsf{tmp}(x)$ in its antecedent, a natural approach to prove it is to perform mathematical induction on $\mathsf{tmp}(x)$. In particular, E is first recorded as an induction hypothesis (IH), and then the predicate $\mathsf{tmp}(x)$ is analyzed in each case of its definition, via a method called *unfolding*, to derive the following entailments E_1, E_2, and E_3:

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

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

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

$$\begin{array}{c l}
\hline
\mathit{true} \vdash \exists y,z,w.\,(u{=}z \wedge v{=}w \wedge t{=}y) & (\vdash_\pi)\text{: Valid, proved by external prover, e.g., Z3.} \\
\hline
\mathsf{ls}(v,t) \vdash \exists y,z,w.\,(\mathsf{ls}(w,y) \wedge u{=}z \wedge v{=}w) & (*\mathsf{P})\text{: Match and remove predicates } \mathsf{ls}(v,t) \text{ and } \mathsf{ls}(w,y). \\
\hline
u{\mapsto}v * \mathsf{ls}(v,t) \vdash \exists y,z,w.\,(z{\mapsto}w * \mathsf{ls}(w,y) \wedge u{=}z) & (*{\mapsto})\text{: Match and remove singleton heaps } u{\mapsto}v \text{ and } z{\mapsto}w. \\
\hline
u{\mapsto}v * \mathsf{ls}(v,t) \vdash \exists y,z.\,(\mathsf{ls}(z,y) \wedge u{=}z) & (\mathsf{P}_{\mathrm{R}})\text{: Unfold } \mathsf{ls}(z,y) \text{ by its inductive case.} \\
\hline
x{\mapsto}u * u{\mapsto}v * \mathsf{ls}(v,t) \vdash \exists y,z.\,(x{\mapsto}z * \mathsf{ls}(z,y)) & (*{\mapsto})\text{: Match and remove singleton heaps } x{\mapsto}u \text{ and } x{\mapsto}z. \\
\hline
x{\mapsto}u * u{\mapsto}v * \mathsf{ls}(v,t) \vdash \exists y.\,\mathsf{ls}(x,y) & (\mathsf{P}_{\mathrm{R}})\text{: Unfold } \mathsf{ls}(x,y) \text{ by inductive case.} \\
\hline
(E_3)\ x{\mapsto}u * u{\mapsto}v * \mathsf{tmp}(v) \vdash \exists y.\,\mathsf{ls}(x,y) & (\mathsf{IH})\text{: Apply IH } E \text{ with subst. } [v/x] \text{, rename } y \text{ to a fresh var } t
\end{array}$$

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

$$\begin{array}{c l}
\hline
\mathit{true} \vdash \exists y.\,y = z & (\vdash_\pi)\text{: Valid, proved by external provers, e.g., Z3.} \\
\hline
\mathsf{ls}(x,z) \vdash \exists y.\,\mathsf{ls}(x,y) & (*\mathsf{P})\text{: Remove predicates } \mathsf{ls}(x,z) \text{ and } \mathsf{ls}(x,y). \\
\hline
x{\mapsto}u * \mathsf{ls}(u,r) \vdash \exists y.\,\mathsf{ls}(x,y) & (\mathsf{IH})\text{: Apply } E_4 \text{ with subst. } [r/t] \text{, and rename } y \text{ to } z. \\
\hline
(E_3)\ x{\mapsto}u * u{\mapsto}v * \mathsf{tmp}(v) \vdash \exists y.\,\mathsf{ls}(x,y) & (\mathsf{IH})\text{: Apply } E_2 \text{ with subst. } [u/x, v/u] \text{, and rename } y \text{ to } r.
\end{array}$$

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

The entailment E_1 can be easily proved by unfolding the predicate $\mathsf{ls}(x, y)$ by its first definition 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 detailed proof tree is depicted in Fig. 2. We can also prove E_3 by the same method, i.e., by 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, the 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, which is demonstrated in Fig. 4, is even simpler than its original proof shown in Fig. 3. The proving process with this new strategy, 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 proved *valid*: its proof tree does not rely on any external hypotheses (Fig. 2). However, when applying E_2 to prove E_3 (Fig. 4), it is not straightforward to conclude about E_3's validity, since the E_2's validity is still *unknown*. This is because the proof of E_2 uses E as a hypothesis (Fig. 2). However, E's validity also relies on the validity of both E_2 and E_3. Therefore, E, E_2 and E_3 jointly form a *mutual induction* proof, in which they are used to prove each other. We will discuss in detail the theoretical principle of this proof technique in Sect. 4.

3. The symbolic-heap separation logic fragment

In this work, we focus on proving entailments in the symbolic-heap fragment of separation logic with inductive heap predicates. Each formula in this fragment consists of a *heap* part symbolically modeling the memory state of a program, and a *pure* part capturing Boolean constraints on the program's variables. This fragment is actively studied in both academia [BCO04, BCO05, IRS13, Bro+16] and industry [BCI11, Cal+15]. We extend this symbolic-heap separation logic fragment with user-defined inductive heap predicates and linear arithmetic constraints to capture more properties of data structures, such as the number of elements, or constraints on these elements' values. We denote our logic fragment as $\mathsf{SL}_{\mathsf{ID}}$, where the subscript ID indicates the support of $\underline{\text{i}}\text{n}\underline{\text{d}}$uctive heap predicates. Its syntax and semantics are introduced below.

Syntax. Figure 5 presents the syntax of formulas in our logic fragment $\mathsf{SL}_{\mathsf{ID}}$. We write x to denote either an *integer variable* representing a numeric property of data structures or a *spatial variable* modeling a memory address. In addition, c represents an *integer constant*, and e is an *integer expression*, which is composed of other expressions by the standard linear arithmetic operators: addition, subtraction, and multiplication by a constant $(+, -, \cdot)$. Moreover, nil is a special *spatial constant* indicating a dangling memory address and a is a *spatial expression* representing a memory address in general. We use π to indicate a *pure atomic formula* which can be an equality constraint among spatial expressions $(=, \neq)$ or a linear arithmetic constraint among integer expressions $(=, \neq, <, \leq, >, \geq)$. These pure atomic formulas compose a *pure formula* Π via application of standard logical operators $(\wedge, \vee, \neg, \rightarrow)$ and standard logical quantification $(\forall, \exists)$ as in first-order logic.

On the other hand, we write σ to denote a *spatial atomic formula*, which can be either a predicate emp, a *singleton heap predicate* $x \overset{\iota}{\mapsto} x_1, \ldots, x_n$, or an *inductive heap predicate* $\mathsf{P}(x_1, \ldots, x_n)$. More specifically, emp models an *empty* memory, i.e., an unallocated memory. The *singleton heap predicate* $x \overset{\iota}{\mapsto} x_1, \ldots, x_n$ describes an n-field data structure of sort ι, which is located in the memory pointed to by x, and has $x_1, \ldots, x_n$ as values of its fields. The sort ι represents a unique type of data structure. This sort notation is omitted when there is only one considered data type in the context, like the motivating example in Sect. 2. Lastly, the *inductive heap predicate* $\mathsf{P}(x_1, \ldots, x_n)$ represents a recursive data structure whose properties are captured by the parameters $x_1, \ldots, x_n$. We will formally define the inductive heap predicate in Definition 3.1. Moreover, these spatial atomic formulas constitute *spatial formula* Σ via the separating conjunction operator $(*)$.

Finally, we write F to represent a *symbolic-heap formula*, whose components are separated by the conjunction connective $(\wedge)$ into two parts: a spatial part represented by Σ and a pure part indicated by Π. One of these two parts can be omitted if Σ is emp or Π is true. We also allow existential quantification $(\exists)$ over both spatial and integer variables of the symbolic-heap formula F.

Definition 3.1 (*Inductive heap predicates* A system of k inductive heap predicates P_i of arity n_i, with $i = 1, \ldots, k$, is defined as:

$$\left\{ \mathsf{P}_i(x_1^i, \ldots, x_{n_i}^i) \overset{\text{def}}{=} F_1^i(x_1^i, \ldots, x_{n_i}^i) \vee \ldots \vee F_{m_i}^i(x_1^i, \ldots, x_{n_i}^i) \right\}_{i=1}^{k}$$

where, each formula $F_j^i(x_1^i, \ldots, x_{n_i}^i)$ is called a *definition case* of $\mathsf{P}_i(x_1^i, \ldots, x_{n_i}^i)$, and this relation is denoted by $F_j^i(x_1^i, \ldots, x_{n_i}^i) \overset{\text{def}}{\Rightarrow} \mathsf{P}_i(x_1^i, \ldots, x_{n_i}^i)$. Moreover, $F_j^i(x_1^i, \ldots, x_{n_i}^i)$ is a *base case*, if it does not contain any predicates mutually defined with $\mathsf{P}_i(x_1^i, \ldots, x_{n_i}^i)$; otherwise, it is an *inductive case*.

Example 3.1 The two predicates $\mathsf{tmp}(x)$ and $\mathsf{ls}(x, y)$ in Sect. 2 are two examples of inductive heap predicates. They are self-recursively defined, i.e., their definitions are only built up from heap predicates of the same symbol, and not from any other inductive heap predicate symbols.

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

Fig. 5. Syntax of $\mathsf{SL_{ID}}$ formulas

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

Fig. 6. Semantics of $\mathsf{SL_{ID}}$ formulas

Example 3.2 The two inductive heap predicates $\mathsf{ListO}(x, y)$ and $\mathsf{ListE}(x, y)$ below were introduced by Brotherston et al. [BDP11] to model segments of linked list data structures which respectively contain *even* and *odd* number of elements. These two predicates are *mutually recursively defined* since the predicate symbols ListO and ListE both appear in the recursive definition of each other.

$$\begin{array}{lll}
\mathsf{ListO}(x, y) & \overset{\text{def}}{=} & x \mapsto y \ \vee\ \exists\, u.(x \mapsto u * \mathsf{ListE}(u, y)) \\
\mathsf{ListE}(x, y) & \overset{\text{def}}{=} & \exists\, u.(x \mapsto u * \mathsf{ListO}(u, y))
\end{array}$$

Semantics. Figure 6 exhibits the semantics of formulas in our separation logic fragment $\mathsf{SL_{ID}}$. We write `Var` to denote a set of variables, `Sort` to represent a set of sorts, and `Val` to indicate a set of values. Moreover, `Loc` is a set of memory addresses (`Loc` $\subset$ `Val`) and `Val`$^+$ is an n-fold Cartesian product of `Val`, where $n{\geq}1$. A model s, h of an $\mathsf{SL_{ID}}$ formula consists of: the *stack* model s, which is a function s: `Var` $\rightarrow$ `Val`, and the *heap* model h, which is a partial function h: (`Loc` $\times$ `Sort`) $\rightharpoonup$ `Val`$^+$.

We write $[\![\Pi]\!]_s$ and $[\![e]\!]_s$ to denote the valuation of a pure formula Π and an expression e under the stack model s. Moreover, $\mathrm{dom}(h)$ denotes the domain of h; $h \# h'$ indicates that h and h' have disjoint domains, i.e., $\mathrm{dom}(h) \cap \mathrm{dom}(h') = \varnothing$; $h \circ h'$ is the union of two disjoint heap models h and h'; and $[f \mid x : y]$ is a function like f except that it returns y for the input x. We follow Reynolds' semantics to consider the *finite* heap models, i.e., $\forall\, h.\ | h | < +\infty$, where $| h |$ is the domain size of h [Rey08].

Entailments. Given the syntax and the semantics of separation logic formulas, we are ready to define separation logic entailments as follows.

Definition 3.2 (*Entailments*) An entailment E between two formulas F_1 and F_2, denoted as $E \triangleq F_1 \vdash F_2$, or just $F_1 \vdash F_2$, is said to be *valid*, iff for all model s, h if s, h satisfies F_1 then it also satisfies F_2. Formally:

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

In the above definition, F_1, F_2 are respectively called the antecedent and the consequent of the entailment.

Definition 3.3 (*Model and counter-model of entailments*) A separation logic model s, h is called a *model* of an entailment $F \vdash G$, iff $s, h \models F$ implies $s, h \models G$. On the contrary, s, h is called a *counter-model* of $F \vdash G$, 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 $E_1, \ldots, E_n$, we write $s, h \models E_1, \ldots, E_n$ if s, h is a model of *all* entailments $E_1, \ldots, E_n$, and $s, h \not\models E_1, \ldots, E_n$ if s, h is a counter-model of *at least one* entailment in $E_1, \ldots, E_n$.

Substitution. We write $[e_1/v_1, \ldots, e_n/v_n]$ to denote a *simultaneous substitution* and $F[e_1/v_1, \ldots, e_n/v_n]$ denotes a formula that is obtained from F by simultaneously replacing all occurrences of the free variables $v_1, \ldots, v_n$ in F by $e_1, \ldots, e_n$, respectively. The simultaneous substitution has the following properties.

Proposition 3.1 (Substitution law for formulas [Rey08]) Consider a formula F, a substitution $\theta = [e_1/v_1, \ldots, e_n/v_n]$, and a separation logic model s, h. Let $s' = [s \mid v_1 : [\![e_1]\!]_s \mid \ldots \mid v_n : [\![e_n]\!]_s]$. Then,

$$s, h \models F\theta, \text{ iff } s', h \models F$$

Theorem 3.1 (Substitution law for entailments) Consider an entailment $F_1 \vdash F_2$ and a substitution θ. If $F_1 \vdash F_2$ is valid, then $F_1\theta \vdash F_2\theta$ is also valid.

Proof. Suppose that $\theta = [e_1/v_1, \ldots, e_n/v_n]$. Consider an arbitrary model s, h of $F_1\theta$, i.e., $s, h \models F_1\theta$. Let $s' = [s \mid v_1 : [\![e_1]\!]_s \mid \ldots \mid v_n : [\![e_n]\!]_s]$. Then by Proposition 3.1, $s, h \models F_1\theta$ implies that $s', h \models F_1$. In addition, the theorem's hypothesis provides that $F_1 \vdash F_2$ is valid, therefore $s', h \models F_2$. It is implied by Proposition 3.1 again that $s, h \models F_2\theta$. Given the hypothesis that $F_1 \vdash F_2$ is valid, we have shown that if s, h is a model of $F_1\theta$, then it is also a model of $F_2\theta$. Since s, h is chosen arbitrarily, by Definition 3.2, the entailment $F_1\theta \vdash F_2\theta$ is valid. □

Syntactic equivalence. We also introduce a new concept of *syntactic equivalence* between two separation logic formulas. This concept will be used in next sections to develop our formal entailment proof system.

Definition 3.4 (*Syntactic equivalence*) The syntactical equivalence relation of two spatial formulas Σ_1 and Σ_2, denoted as $\Sigma_1 \cong \Sigma_2$, is recursively defined as follows:

(1) $\mathsf{emp} \cong \mathsf{emp}$ (2) $x \overset{\iota}{\mapsto} x_1, \ldots, x_n \cong x \overset{\iota}{\mapsto} x_1, \ldots, x_n$ (3) $\mathsf{P}(x_1, \ldots, x_n) \cong \mathsf{P}(x_1, \ldots, x_n)$
(4) $(\Sigma_1 \cong \Sigma_1') \wedge (\Sigma_2 \cong \Sigma_2') \rightarrow (\Sigma_1 * \Sigma_2 \cong \Sigma_1' * \Sigma_2') \wedge (\Sigma_1 * \Sigma_2 \cong \Sigma_2' * \Sigma_1')$

4. A mutual induction principle for entailment proof

In this section, we first introduce a general schema of Noetherian induction, a.k.a., well-founded induction, and apply it to prove separation logic entailments.

Noetherian induction [Bun01]. Given a conjecture $\mathcal{P}(\alpha)$, with α is a structure of type τ. The general schema of Noetherian induction states that for an arbitrary structure α, if $\mathcal{P}$ holds for all sub-structure β of α implies that $\mathcal{P}$ also holds for α, then $\mathcal{P}$ holds for all structure α.

Formally:

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

where $\prec_\tau$ is a well-founded relation on τ, i.e., there is no infinite descending chain, like $\ldots \prec_\tau \alpha_n \prec_\tau \ldots \prec_\tau \alpha_2 \prec_\tau \alpha_1$. 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 separation logic entailments using Noetherian induction. We observe that in Reynolds' semantics, the heap domain of any separation model s, h is finite, i.e., $| h | < +\infty$ [Rey08]. This property inspires us to define a relation among separation logic models by comparing the size of their heap domains. We call this relation as the *model order*, and it is a *well-founded* relation.

To prove a separation logic entailment by using this model order, we will show that for an arbitrary separation logic model, if all the smaller models satisfying the entailment implies that the given model also satisfies it, then the entailment is satisfied by all separation logic models and is valid. We formally describe the model order and the induction principle as follows.

Definition 4.1 (*Model order*) The *model order* is a binary relation between separation logic models. It is denoted by $\prec$, and is defined as: $s_1, h_1 \prec s_2, h_2$, iff $| h_1 | < | h_2 |$.

Theorem 4.1 (Well-founded relation) The model order $\prec$ is a well-founded relation.

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

Theorem 4.2 (Induction principle) Consider a separation logic entailment E. For all model s, h, if all smaller models s', h' satisfy E implies that s, h also satisfies E, then E is valid. 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 separation logic 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 others. In the following, we will introduce our *mutual induction principle*. Note that the induction principle in Theorem 4.2 is an instance of this principle when only one entailment takes part in the induction proof.

Theorem 4.3 (Mutual induction principle) Given n entailments $E_1, \ldots, E_n$. For all model s, h, if all smaller models s', h' satisfy $E_1, \ldots, E_n$ implies that s, h also satisfies $E_1, \ldots, E_n$, then all the entailments $E_1, \ldots, E_n$ are valid. Formally:

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

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

5. The mutual induction proof system

In this section, we formally describe a proof system, which implements the mutual induction principle to prove separation logic entailments. The proof system comprises logical rules dealing with the logical structures of entailments in Sect. 5.1, and induction rules recording and applying induction hypotheses in Sect. 5.2. We also introduce a general proof search procedure and discuss its soundness in Sects. 5.3 and 5.4.

Each inference rule in our proof system has zero or more premises, one conclusion, and possibly one side condition. Each of the premises or the conclusion is described in the same form of a triple $\mathcal{H},\ \rho,\ F_1 \vdash F_2$. More specifically, $F_1 \vdash F_2$ is an entailment. $\mathcal{H}$ is a set of entailments serving as hypotheses and their validity statuses, where each hypothesis is annotated with either the symbol ✓ or the symbol **?** indicating that the hypothesis is currently *valid* or *unknown*. These hypotheses are recorded during a proof search and can be used to prove the entailment $F_1 \vdash F_2$. Lastly, ρ is a proof trace capturing a chronological list of inference rules applied by the proof search procedure to reach $F_1 \vdash F_2$.

We also call the entailment in each rule's conclusion as the goal entailment. The rules with empty premise are called the 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 applied by the proof search procedure. We utilize basic operations to manipulate proof traces, which include proof trace insertion: $(R) :: \rho$, membership checking: $(R) \in \rho$, and proof trace concatenation: $\rho_1 \mathbin{@} \rho_2$.

5.1. Logical rules

Figure 7 presents logical rules of our proof system to deal with the logical structure of separation logic entailments. For brevity, we write $\vec{x}$ to indicate a list of variables $x_1, \ldots, x_n$, and $|\ \vec{x}\ |$ means the number of variables in $\vec{x}$. We denote the symbolic-heap formula $\exists\,\vec{x}.(\Sigma \wedge \Pi)$ as F, where the variable list $\vec{x}$ can be empty. When $F \equiv \Sigma \wedge \Pi$, we write $F * \Sigma'$ to denote $\Sigma * \Sigma' \wedge \Pi$ and $F \wedge \Pi'$ to denote $\Sigma \wedge \Pi \wedge \Pi'$. When $\vec{u}$ and $\vec{v}$ are two variable lists of the same length, i.e., $\vec{u} = u_1, \ldots, u_n$ and $\vec{v} = v_1, \ldots, v_n$, we write $\vec{u}{=}\vec{v}$ to indicate $(u_1{=}v_1) \wedge \ldots \wedge (u_n{=}v_n)$. We also write $\vec{x} \mathbin{\#} \vec{y}$ to denote $\vec{x}$ and $\vec{y}$ are disjoint, i.e., $\nexists u.(u \in \vec{x} \wedge u \in \vec{y})$. Finally, $\mathtt{FV}(F)$ is a set of free variables appearing in the formula F.

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

- *Axiom rules*. The rule $\vdash_\pi$ proves a pure entailment $\Pi_1 \vdash \Pi_2$ by invoking off-the-shelf provers such as Z3 [MB08] to check the implication $\Pi_1 \rightarrow \Pi_2$ in its side condition. The two rules $\bot^\pi_{\mathrm{L}}$ and $\bot^\sigma_{\mathrm{L}}$ decide an entailment *vacuously* valid if its antecedent is unsatisfiable, i.e., the antecedent contains a contradiction (as indicated in the rule $\bot^\pi_{\mathrm{L}}$'s side condition $\Pi \rightarrow \mathsf{false}$) or separating singleton heap predicates having the same memory address ($u \overset{\iota_1}{\mapsto} \vec{v} * u \overset{\iota_2}{\mapsto} \vec{w}$, which violates the separating condition of separation logic).
- *Normalization rules*. These rules simplify their goal entailments by either removing equalities ($=_{\mathrm{L}}$), eliminating existentially quantified variables ($\exists_{\mathrm{L}}, \exists_{\mathrm{R}}$), or the empty heap predicate $\mathtt{emp}$ ($\mathsf{E}_{\mathrm{L}}, \mathsf{E}_{\mathrm{R}}$) from antecedents or consequents of the entailments.
- *Matching rules*. The two rules $*{\mapsto}$ and $*\mathsf{P}$ match and remove *identical* spatial atoms from two sides of their goal entailments. More specifically, the matching condition, i.e., the identical condition, is guaranteed by adding equality constraints on these spatial atoms' arguments into the consequents of the derived entailments. These constraints need to be proved later during the proof derivation.
- *Unfolding rule*. The rule P_{R} derives new entailments by unfolding a heap predicate in the consequent of a goal entailment by its inductive definition. The original goal entailment is valid if at least one of the derived entailments is valid. On the other hand, heap predicates in the entailment's antecedent will be handled by the induction rule ID in the next subsection.

$$\vdash_\pi \; \frac{}{\mathcal{H},\, \rho,\, \Pi_1 \vdash \Pi_2}\; \Pi_1 \rightarrow \Pi_2 \qquad \bot^\pi_{\mathrm{L}} \; \frac{}{\mathcal{H},\, \rho,\, F_1 \wedge \Pi \vdash F_2}\; \Pi \rightarrow \mathsf{false} \qquad \bot^\sigma_{\mathrm{L}} \; \frac{}{\mathcal{H},\, \rho,\, F_1 * u \xmapsto{\iota_1} \vec{v} * u \xmapsto{\iota_2} \vec{w} \vdash F_2}$$

$$\exists_{\mathrm{L}} \; \frac{\mathcal{H},\, \rho',\, F_1[u/x] \vdash F_2}{\mathcal{H},\, \rho,\, \exists x.F_1 \vdash F_2}\; u \notin \mathtt{FV}(F_2) \qquad \exists_{\mathrm{R}} \; \frac{\mathcal{H},\; \rho',\; F_1 \vdash \exists \vec{x}.F_2[u/v]}{\mathcal{H},\; \rho,\; F_1 \vdash \exists \vec{x}, v.(F_2 \wedge u{=}v)}$$

$$=_{\mathrm{L}} \; \frac{\mathcal{H},\; \rho',\; F_1[u/v] \vdash F_2[u/v]}{\mathcal{H},\; \rho,\; F_1 \wedge u{=}v \vdash F_2} \qquad \mathsf{P}_{\mathrm{R}} \; \frac{\mathcal{H},\; \rho',\; F_1 \vdash \exists \vec{x}.(F_2 * F^{\mathsf{P}}_i(\vec{u}))}{\mathcal{H},\; \rho,\; F_1 \vdash \exists \vec{x}.(F_2 * \mathsf{P}(\vec{u}))}\; F^{\mathsf{P}}_i(\vec{u}) \overset{\text{def}}{\Rightarrow} \mathsf{P}(\vec{u})$$

$$\mathsf{E}_{\mathrm{L}} \; \frac{\mathcal{H},\; \rho',\; F_1 \vdash F_2}{\mathcal{H},\; \rho,\; F_1 * \mathsf{emp} \vdash F_2} \qquad \mathsf{E}_{\mathrm{R}} \; \frac{\mathcal{H},\; \rho',\; F_1 \vdash \exists \vec{x}.F_2}{\mathcal{H},\; \rho,\; F_1 \vdash \exists \vec{x}.(F_2 * \mathsf{emp})}$$

$$*\mathsf{P} \; \frac{\mathcal{H},\; \rho',\; F_1 \vdash \exists \vec{x}.(F_2 \wedge \vec{u}{=}\vec{v})}{\mathcal{H},\; \rho,\; F_1 * \mathsf{P}(\vec{u}) \vdash \exists \vec{x}.(F_2 * \mathsf{P}(\vec{v}))}\; \vec{u} \,\#\, \vec{x} \qquad *\!\mapsto \; \frac{\mathcal{H},\; \rho',\; F_1 \vdash \exists \vec{x}.(F_2 \wedge u{=}t \wedge \vec{v}{=}\vec{w})}{\mathcal{H},\; \rho,\; F_1 * u \xmapsto{\iota} \vec{v} \vdash \exists \vec{x}.(F_2 * t \xmapsto{\iota} \vec{w})}\; u \notin \vec{x}, \vec{v} \,\#\, \vec{x}$$

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

$$\mathsf{ID} \; \frac{\mathcal{H} \cup \{(H, \mathbf{?})\},\, (\mathsf{ID}) :: \rho,\, F_1 * F^{\mathsf{P}}_1(\vec{u}) \vdash F_2 \qquad \ldots \qquad \mathcal{H} \cup \{(H, \mathbf{?})\},\, (\mathsf{ID}) :: \rho,\, F_1 * F^{\mathsf{P}}_m(\vec{u}) \vdash F_2}{\mathcal{H},\; \rho,\; F_1 * \mathsf{P}(\vec{u}) \vdash F_2}\; \dagger_{(\mathsf{ID})}$$

$$\text{with } H \triangleq F_1 * \mathsf{P}(\vec{u}) \vdash F_2, \text{ and } \dagger_{(\mathsf{ID})}\!:\;\; \mathsf{P}(\vec{u}) \overset{\text{def}}{=} F^{\mathsf{P}}_1(\vec{u}) \vee \ldots \vee F^{\mathsf{P}}_m(\vec{u})$$

$$\mathsf{IH} \; \frac{\mathcal{H} \cup \{(H, \mathit{status})\},\;\, (\mathsf{IH}) :: \rho,\;\, F_4\theta * \Sigma' \wedge \Pi_1 \vdash F_2}{\mathcal{H} \cup \{(H \triangleq \Sigma_3{\wedge}\Pi_3{\vdash}F_4, \mathit{status})\},\, \rho,\, \Sigma_1 \wedge \Pi_1 \vdash F_2}\; \exists\theta,\!\Sigma'.(\Sigma_1{\cong}\Sigma_3\theta{*}\Sigma' \wedge \Pi_1{\rightarrow}\Pi_3\theta);\;\; \dagger_{(\mathsf{IH})}$$

$$\begin{aligned} \text{with } \dagger_{(\mathsf{IH})}\!:\quad (\mathit{status} = \checkmark)\quad &\vee\;\; \exists\, \iota, u, \vec{v}, \Sigma''.(\Sigma' \cong u \xmapsto{\iota} \vec{v} * \Sigma'') \\ &\vee\;\; \exists\, \rho_1, \rho_2.(\rho = \rho_1 @ [(*\!\mapsto)] @ \rho_2 \wedge (\mathsf{ID}) \notin \rho_1 \wedge (\mathsf{ID}) \in \rho_2). \end{aligned}$$

Fig. 8. Induction rules

5.2. Induction rules

Figure 8 presents two induction rules ID and IH, which are the main rules implementing our mutual induction principle. We will carefully explain them below.

The rule ID 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 search procedure is able to prove it valid. Generally, an entailment E can be concluded valid if the proof search procedure is able to derive a proof tree $\mathcal{T}$, where (i) every leaf of $\mathcal{T}$ is empty via applications of axiom rules, and (ii) all hypotheses used by the rule IH must be derived within $\mathcal{T}$.

The rule IH applies an appropriate hypothesis $H \triangleq \Sigma_3 \wedge \Pi_3 \vdash F_4$ to prove its goal entailment $E \triangleq \Sigma_1 \wedge \Pi_1 \vdash F_2$. It firstly matches[1] the antecedents of H and E by a substitution θ, i.e., there exists a spatial formula 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 $(\Sigma_1 \wedge \Pi_1) \vdash (\Sigma_3\theta * \Sigma' \wedge \Pi_3\theta \wedge \Pi_1) \vdash (F_4\theta * \Sigma' \wedge \Pi_1)$. Here, we apply the substitution law for entailments (Theorem 3.1) to obtain the entailment $\Sigma_3\theta \wedge \Pi_3\theta \vdash F_4\theta$ from the hypothesis H. The proof system then derives a new sub-goal entailment $F_4\theta * \Sigma' \wedge \Pi_1 \vdash F_2$ as shown in the premise of the rule IH. Details of this proof derivation can be referred to in the soundness proof of the rule IH in Sect. 5.4.

[1] In an earlier work [Ta+16] of this paper, we used the term "unify" to describe the induction hypothesis application. However, the use of "unify" in this context is imprecise, and the more suitable term is "match", since *matching* is a special version of *unification*, which allows substitution on only the first of each pair of formulas or terms [KN87, Har09]. Note that this *matching* is unrelated to the *matching* inference rules which are commonly used in separation logic literature.

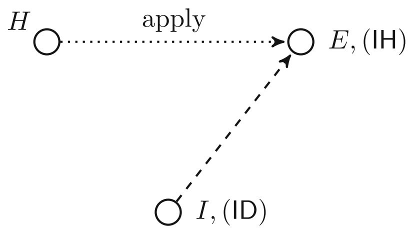

Fig. 9. Applying hypothesis

The side condition $\dagger_{(\mathsf{IH})}$ of the rule IH ensures the decreasing condition of our mutual induction principle. In particular, suppose that the proof search procedure applies a hypothesis H in $\mathcal{H}$ to prove an entailment E by using the rule IH. If the status of H is ✓, which is denoted by the first condition in $\dagger_{(\mathsf{IH})}$, then H is already proved to be valid; thus it can be surely used to prove other entailments. Otherwise, the status of H is **?**, and H may participate in a mutual induction proof with an entailment I in the proof path of E. We illustrate this induction hypothesis application in Fig. 9, where the entailment I has been recorded earlier as an induction hypothesis by an application of the rule ID.

In the latter case, our mutual induction principle requires the decrease of the model size when applying the hypothesis H to prove the entailment I. We will show that this decreasing condition holds if at least one of the last two conditions of $\dagger_{(\mathsf{IH})}$ is satisfied. For the convenience of presentation, we first restate these two conditions of $\dagger_{(\mathsf{IH})}$ as follows:

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

(ii) $\exists\, \rho_1, \rho_2.(\rho = \rho_1@[(*\mapsto)]@\rho_2 \;\wedge\; (\mathsf{ID}) \notin \rho_1 \;\wedge\; (\mathsf{ID}) \in \rho_2)$, which requires that there is a removal step of a singleton heap predicate, which is applied by the rule $*\mapsto$ in a proof path from the most recent induction step ID to the current hypothesis application IH.

Now, to prove the decreasing condition when applying the hypothesis H to prove the entailment I, we will consider an arbitrary model s, h of 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 $|\ h_e\ | \le |\ h\ |$ since applications of all inference rules never increase heap model size of corresponding entailments. Moreover, when applying H to prove E, the model s', h' of H, which corresponds to s_e, h_e of E, satisfies $|\ h'\ | \le |\ h_e\ |$, due to the matching step in rule IH. We consider two following cases. If the condition (i) is satisfied, then the size of heap model 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 the condition (ii) is satisfied, then $|\ h_e\ | < |\ h\ |$ since there is a singleton heap predicate, whose heap model size is 1, is removed when deriving I to E. This implies that $|\ h'\ | < |\ h\ |$. In summary, we obtain that $|\ h'\ | < |\ h\ |$ for both the two cases; thus, $s', h' \prec s, h$. This concludes our explanation about the rule IH.

5.3. A general proof search procedure

Figure 10 presents our proof search procedure Prove, which is designed in a self-recursive manner. Its inputs consist of a set of hypotheses $\mathcal{H}$, a proof trace ρ, and an entailment $F \vdash G$. These inputs correspond to the components of the conclusions of all inference rules. When Prove is invoked for the first time to prove a goal entailment, the induction hypothesis set $\mathcal{H}$ and the proof trace ρ are initially assigned to empty ($\varnothing$ and [ ]). Their values will be accumulated later during the proof derivation.

Generally, to prove the input entailment $F \vdash G$, the procedure Prove firstly discovers among all inference rules presented in Figs. 7 and 8, a set of suitable rules $\mathcal{R}$ whose conclusions can be unified with the entailment $F \vdash G$. This inference rule discovery is done by invoking an auxiliary procedure FindRules at line 1. If no suitable rule is found, the proof search procedure immediately returns `UNKN`, indicating that it is unable to prove the entailment (line 2).

```
Procedure Prove($\mathcal{H}, \rho, F \vdash G$)
Input: $\mathcal{H}$, $\rho$ and $F \vdash G$ are respectively a set of hypotheses, a proof trace, and a goal entailment.
Output: Validity result (VALID or UNKN), a set of derived entailments with their validity statuses, and a set
of hypotheses used in the proof of $F \vdash G$.
1:  $\mathcal{R} \leftarrow$ FindRules($\mathcal{H}, \rho, F \vdash G$)
2:  if $\mathcal{R} = \varnothing$ then return (UNKN, $\varnothing, \varnothing$)                // no rule is selected
3:  for each R in $\mathcal{R}$ do
4:      if R $\in \{\vdash_\pi, \bot^\pi_{\mathrm{L}}, \bot^\sigma_{\mathrm{L}}\}$ then return (VALID, $\varnothing, \varnothing$)      // R is an axiom rule
5:      $res \leftarrow$ VALID
6:      $\mathcal{H}_{\text{used}} \leftarrow \varnothing$
7:      $\mathcal{H}_{\text{drvd}} \leftarrow \varnothing$
8:      if R = IH with hypothesis $H$ then $\mathcal{H}_{\text{used}} \leftarrow \mathcal{H}_{\text{used}} \cup \{H\}$
9:      for each ($\mathcal{H}_i, \rho_i, F_i \vdash G_i$) in GetPremises(R) do
10:         $res, \mathcal{H}_{\text{drvd}}, \mathcal{H}'_{\text{used}} \leftarrow$ Prove($\mathcal{H}_i \oplus \mathcal{H}_{\text{drvd}}, \rho_i, F_i \vdash G_i$)
11:         if $res$ = UNKN then break
12:         $\mathcal{H}_{\text{used}} \leftarrow \mathcal{H}_{\text{used}} \cup \mathcal{H}'_{\text{used}}$
13:     if $res$ = UNKN then continue
14:     else if $\mathcal{H}_{\text{used}} \subseteq$ (GetHypotheses($\mathcal{H}_{\text{drvd}}$) $\cup \{F \vdash G\}$) then $\mathcal{H}_{\text{drvd}} \leftarrow \mathcal{H}_{\text{drvd}} \oplus \{(F \vdash G, \checkmark)\}$
15:     else $\mathcal{H}_{\text{drvd}} \leftarrow \mathcal{H}_{\text{drvd}} \oplus \{(F \vdash G, \mathbf{?})\}$
16:     return (VALID, $\mathcal{H}_{\text{drvd}}, \mathcal{H}_{\text{used}}$)                // all derived premises are proved
17: return (UNKN, $\varnothing, \varnothing$)                           // all rules fail to prove $F \vdash G$
```

Fig. 10. The mutual induction proof search procedure

Otherwise, it subsequently processes each discovered rule R in $\mathcal{R}$ and either (i) returns VALID to announce a valid result, if an axiom rule is selected (line 4), or (ii) recursively searches for proofs of new sub-goal entailments derived in the premises of R (lines 9–16). In the latter case, the procedure switches to apply the next discovered rule if one of the currently derived sub-goal entailments is not proved (line 13), or returns VALID if all of them are proved by the current rule R (line 16). Finally, it simply returns UNKN when it cannot prove the goal entailment with all selected rules R (line 17).

The procedure Prove uses a local variable $\mathcal{H}_{\text{used}}$ to store all hypotheses used during the proof search. $\mathcal{H}_{\text{used}}$ is updated when the rule IH is applied (line 8) or after the procedure finishes proving a derived entailment (lines 10 and 12). Prove also uses another variable $\mathcal{H}_{\text{drvd}}$ to capture all entailments derived during the proof search and 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 14). Otherwise, the entailment may participate in a mutual induction proof, thus its status is assigned to *unknown* **?** (line 15).

At line 10, the procedure Prove uses not only the hypothesis set $\mathcal{H}_i$, introduced by the selected inference rule, but also the set $\mathcal{H}_{\text{drvd}}$ containing entailments derived during the proof search to prove a new goal entailment $F_i \vdash G_i$. This hypothesis application reflects our mutual induction principle which allows a goal entailment and other derived entailments to be used as hypotheses to prove each other. Lastly, we use the *union and update* operator $\oplus$ in the algorithm to insert new entailments and their statuses into a set of hypotheses or to update existing entailments with their new statuses.

5.4. Soundness and completeness

Our mutual induction proof system is *sound* but *incomplete*. If it can prove an entailment, then the entailment is semantically valid. Otherwise, it cannot conclude about the validity of the entailment. The proof system's soundness is formally stated in the following Theorem 5.1. Moreover, the incompleteness of our proof system is an unavoidable drawback of any proof system for logic fragments containing inductive predicates and linear arithmetic. In essence, these logic fragments can represent any constraints in Peano arithmetic, e.g., addition and multiplication. According to Gödel's second incompleteness theorem [God92], every consistent axiomatic system which includes Peano arithmetic cannot prove its own consistency. Consequently, there does not exist any proof system which can either prove or disprove an arbitrary entailment. We will discuss more about this incompleteness when explaining the limitation of our proof system in Sect. 9.

Theorem 5.1 (Soundness) Given an entailment E, if the proof search procedure Prove returns `VALID` when proving E using an empty set of hypotheses ($\mathcal{H} = \varnothing$), then E is semantically valid.

Proof. We first prove that all the inference rules utilized by our proof system (Figs. 7 and 8) are sound. More specifically, we will show that if the entailments in the premises of these rules are valid, and their side conditions hold, then the goal entailments in their conclusions are also valid. After that, we will argue that when the proof search procedure (Fig. 10) applies these inference rules to prove an entailment, the corresponding proof tree reflects accurately the mutual induction principle. The details are as follows. □

The soundness of the inference rules:
- Axiom rules $\bot^{\pi}_{\mathrm{L}}$, $\bot^{\sigma}_{\mathrm{L}}$ and $\vdash_{\pi}$:

$$\bot^{\pi}_{\mathrm{L}}\ \frac{}{\mathcal{H},\ \rho,\ F_1 \wedge \Pi \vdash F_2}\ \Pi \rightarrow \mathsf{false} \qquad \bot^{\sigma}_{\mathrm{L}}\ \frac{}{\mathcal{H},\ \rho,\ F_1 * u \overset{\iota_1}{\mapsto} \vec{v} * u \overset{\iota_2}{\mapsto} \vec{w} \vdash F_2} \qquad \vdash_{\pi}\ \frac{}{\mathcal{H},\ \rho,\ \Pi_1 \vdash \Pi_2}\ \Pi_1 \rightarrow \Pi_2$$

For the two rules $\bot^{\pi}_{\mathrm{L}}$, $\bot^{\sigma}_{\mathrm{L}}$, the antecedents of their goal entailments are equivalent to false since they either contain a contradiction ($\Pi \rightarrow \mathsf{false}$ in the rule $\bot^{\pi}_{\mathrm{L}}$) or contain two singleton heaps having the same memory address ($u \overset{\iota_1}{\mapsto} \vec{v} * u \overset{\iota_2}{\mapsto} \vec{w}$ in the rule $\bot^{\sigma}_{\mathrm{L}}$). Therefore, these entailments are valid.

For the rule $\vdash_{\pi}$, since its side condition $\Pi_1 \rightarrow \Pi_2$ holds, it's evident that the goal entailment $\Pi_1 \vdash \Pi_2$ is valid.
- Rules $=_{\mathrm{L}}$, $\exists_{\mathrm{L}}$ and $\exists_{\mathrm{R}}$:

$$=_{\mathrm{L}}\ \frac{\mathcal{H},\ \rho',\ F_1[u/v] \vdash F_2[u/v]}{\mathcal{H},\ \rho,\ F_1 \wedge u{=}v \vdash F_2} \qquad \exists_{\mathrm{L}}\ \frac{\mathcal{H},\ \rho',\ F_1[u/x] \vdash F_2}{\mathcal{H},\ \rho,\ \exists x.F_1 \vdash F_2}\ u \notin \mathtt{FV}(F_2) \qquad \exists_{\mathrm{R}}\ \frac{\mathcal{H},\ \rho',\ F_1 \vdash \exists \vec{x}.F_2[u/v]}{\mathcal{H},\ \rho,\ F_1 \vdash \exists \vec{x}, v.(F_2 \wedge u{=}v)}$$

For the rule $=_{\mathrm{L}}$, consider an arbitrary model s, h such that $s, h \models F_1 \wedge u{=}v$. It follows that $s(u) = s(v)$, therefore $s, h \models F_1[u/v]$. Since the entailment $F_1[u/v] \vdash F_2[u/v]$ in this rule's premise is valid, $s, h \models F_2[u/v]$. Recall that $s(u) = s(v)$, hence $s, h \models F_2$. Since s, h is chosen arbitrarily, the entailment $F_1 \wedge u{=}v \vdash F_2$ in this rule's conclusion is valid.

For the rule $\exists_{\mathrm{L}}$, consider an arbitrary model s, h such that $s, h \models \exists x.F_1$. Then the stack model s can be extended with an integer value $i \in \mathtt{Int}$ of x to obtain a new model s', i.e., $s' = [\, s \mid x : i\,]$, such that $s', h \models F_1$. Let s'' be a stack model such that $s'' = [\, s \mid u : i\,]$. Then, $s'(x) = s''(u)$ (both are equal to i) and it follows from $s', h \models F_1$ that $s'', h \models F_1[u/x]$. Since the entailment $F_1[u/x] \vdash F_2$ in this rule's premise is valid, $s'', h \models F_2$, or $[\, s \mid u : i\,] \models F_2$. It follows from this rule's side condition $u \notin \mathtt{FV}(F_2)$ that $s, h \models F_2$. Since s, h is chosen arbitrarily, the goal entailment $\exists x.F_1 \vdash F_2$ in this rule's conclusion is valid.

For the rule $\exists_{\mathrm{R}}$, consider an arbitrary model s, h such that $s, h \models F_1$. Since the entailment $F_1 \vdash \exists \vec{x}.F_2[u/v]$ in this rule's premise is valid, $s, h \models \exists \vec{x}.F_2[u/v]$. It follows that $s, h \models \exists \vec{x}, v.(F_2 \wedge u{=}v)$, because we can always choose $s(u)$ as the value of the existential variable v. Since s, h is chosen arbitrarily, the entailment $F_1 \vdash \exists \vec{x}, v.(F_2 \wedge u{=}v)$ in this rule's conclusion is valid.

- Rules $\mathsf{E_L}$ and $\mathsf{E_R}$:

$$\mathsf{E_L}\ \frac{\mathcal{H},\ \rho',\ F_1 \vdash F_2}{\mathcal{H},\ \rho,\ F_1 * \mathsf{emp} \vdash F_2} \qquad\qquad \mathsf{E_R}\ \frac{\mathcal{H},\ \rho',\ F_1 \vdash \exists\,\vec{x}.F_2}{\mathcal{H},\ \rho,\ F_1 \vdash \exists\,\vec{x}.(F_2 * \mathsf{emp})}$$

It is evident that the two formulas $F_1 * \mathsf{emp}$ and F_1 in the rule $\mathsf{E_L}$ are semantically equivalent. In addition, $F_2 * \mathsf{emp}$ and F_2 in the rule $\mathsf{E_R}$ are also semantically equivalent. Therefore, if the entailments $F_1 \vdash F_2$ and $F_1 \vdash \exists\,\vec{x}.F_2$ in these rules' premises are valid, so are the goal entailments $F_1 * \mathsf{emp} \vdash F_2$ and $F_1 \vdash \exists\,\vec{x}.(F_2 * \mathsf{emp})$ in these rules' conclusions.
- Rules $*\!\mapsto$ and $*\mathsf{P}$:

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

For the rule $*\!\mapsto$, consider an arbitrary model s, h such that $s, h \models F_1 * u \overset{\iota}{\mapsto} \vec{v}$. Then, there exist two heap models h_1 and h_2 such that $h = h_1 \circ h_2$, and $s, h_1 \models F_1$, and $s, h_2 \models u \overset{\iota}{\mapsto} \vec{v}$. Since the entailment $F_1 \vdash \exists\,\vec{x}.(F_2 \wedge u = t \wedge \vec{v} = \vec{w})$ in the premise of this rule is valid, $s, h_1 \models \exists\,\vec{x}.(F_2 \wedge u{=}t \wedge \vec{v}{=}\vec{w})$. Then, s can be extended with some integer values of $\vec{x}$ to obtain a new stack model s' such that $s', h_1 \models F_2 \wedge u{=}t \wedge \vec{v}{=}\vec{w}$. Besides, the side condition of this rule gives $u \notin \vec{x}$ and $\vec{v} \,\#\, \vec{x}$. Recall that s' is extend from s with values of $\vec{x}$, and $s, h_2 \models u \overset{\iota}{\mapsto} \vec{v}$, therefore $s', h_2 \models u \overset{\iota}{\mapsto} \vec{v}$. We have shown that $s', h_1 \models F_2 \wedge u{=}t \wedge \vec{v}{=}\vec{w}$, and $s, h_2 \models u \overset{\iota}{\mapsto} \vec{v}$, and $h = h_1 \circ h_2$. It follows that $s', h \models F_2 * u \overset{\iota}{\mapsto} \vec{v} \wedge u{=}t \wedge \vec{v}{=}\vec{w}$. Hence, $s', h \models F_2 * u \overset{\iota}{\mapsto} \vec{v}$, and $s'(u) = s'(t)$, and $s'(\vec{v}) = s'(\vec{w})$. Consequently, $s', h \models F_2 * t \overset{\iota}{\mapsto} \vec{w}$. Since s' is extended from s with values of $\vec{x}$, it follows from the semantics of existential quantification that $s, h \models \exists\,\vec{x}.(F_2 * t \overset{\iota}{\mapsto} \vec{w})$. Recall that s, h is chosen arbitrarily, hence the entailment $F_1 * u \overset{\iota}{\mapsto} \vec{v} \vdash \exists\,\vec{x}.(F_2 * t \overset{\iota}{\mapsto} \vec{w})$ in this rule's conclusion is valid.

The soundness of the rule $*\mathsf{P}$ can be proved similarly.
- Rule $\mathsf{P_R}$:

$$\mathsf{P_R}\ \frac{\mathcal{H},\ \rho',\ F_1 \vdash \exists\,\vec{x}.(F_2 * F^{\mathsf{P}}_i(\vec{u}))}{\mathcal{H},\ \rho,\ F_1 \vdash \exists\,\vec{x}.(F_2 * \mathsf{P}(\vec{u}))}\ F^{\mathsf{P}}_i(\vec{u}) \overset{\text{def}}{\Rightarrow} \mathsf{P}(\vec{u})$$

Consider an arbitrary model s, h such that $s, h \models F_1$. Since the entailment $F_1 \vdash \exists\,\vec{x}.(F_2 * F^{\mathsf{P}}_i(\vec{u}))$ in the rule's premise is valid, it follows that $s, h \models \exists\,\vec{x}.(F_2 * F^{\mathsf{P}}_i(\vec{u}))$. In addition, the rule's side condition $F^{\mathsf{P}}_i(\vec{u}) \overset{\text{def}}{\Rightarrow} \mathsf{P}(\vec{u})$ indicates that $F^{\mathsf{P}}_i(\vec{u})$ is a definition case of $\mathsf{P}(\vec{u})$. Therefore, $s, h \models \exists\,\vec{x}.(F_2 * \mathsf{P}(\vec{u}))$. Since s, h is chosen arbitrarily, it follows that the entailment $F_1 \vdash \exists\,\vec{x}.(F_2 * \mathsf{P}(\vec{u}))$ in this rule's conclusion is valid.
- Rule ID:

$$\mathsf{ID}\ \frac{\mathcal{H} \cup \{(H, \mathbf{?})\},\ (\mathsf{ID}) :: \rho,\ F_1 * F^{\mathsf{P}}_1(\vec{u}) \vdash F_2 \qquad \dots \qquad \mathcal{H} \cup \{(H, \mathbf{?})\},\ (\mathsf{ID}) :: \rho,\ F_1 * F^{\mathsf{P}}_m(\vec{u}) \vdash F_2}{\mathcal{H},\ \rho,\ F_1 * \mathsf{P}(\vec{u}) \vdash F_2}\ \dagger_{(\mathsf{ID})}$$

with $H \triangleq F_1 * \mathsf{P}(\vec{u}) \vdash F_2$, and $\dagger_{(\mathsf{ID})}$: $\mathsf{P}(\vec{u}) \overset{\text{def}}{=} F^{\mathsf{P}}_1(\vec{u}) \vee \dots \vee F^{\mathsf{P}}_m(\vec{u})$

Consider an arbitrary model s, h such that $s, h \models F_1 * \mathsf{P}(\vec{u})$. According to the side condition $\mathsf{P}(\vec{u}) \overset{\text{def}}{=} F^{\mathsf{P}}_1(\vec{u}) \vee \dots \vee F^{\mathsf{P}}_m(\vec{u})$ of this rule, $F^{\mathsf{P}}_1(\vec{u}), \dots, F^{\mathsf{P}}_m(\vec{u})$ are all definition cases of $\mathsf{P}(\vec{u})$. Since $s, h \models F_1 * \mathsf{P}(\vec{u})$, it follows that $s, h \models F_1 * F^{\mathsf{P}}_i(\vec{u})$, for all $i = 1 \dots m$. On the other hand, $F_1 * F^{\mathsf{P}}_1(\vec{u}), \dots, F_1 * F^{\mathsf{P}}_m(\vec{u})$ are the antecedents of all the entailments $F_1 * F^{\mathsf{P}}_1(\vec{u}) \vdash F_2, \dots, F_1 * F^{\mathsf{P}}_m(\vec{u}) \vdash F_2$ in this rule's premises. These entailments are valid and have the same consequent F_2. Therefore, $s, h \models F_2$. Since s, h is chosen arbitrarily, it follows that the entailment in the rule's conclusion is valid.

- Rule IH:

$$\mathsf{IH}\ \frac{\mathcal{H} \cup \{(H,\, status)\},\ (\mathsf{IH}) :: \rho,\ \ F_4\theta * \Sigma' \wedge \Pi_1 \vdash F_2}{\mathcal{H} \cup \{(H \triangleq \Sigma_3 \wedge \Pi_3 \vdash F_4,\, status)\},\ \rho,\ \ \Sigma_1 \wedge \Pi_1 \vdash F_2}\ \exists\, \theta, \Sigma'.(\Sigma_1 \cong \Sigma_3\theta * \Sigma' \wedge \Pi_1 \rightarrow \Pi_3\theta);\quad \dagger_{(\mathsf{IH})}$$

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

On one hand, the side conditions $\Sigma_1 \cong \Sigma_3\theta * \Sigma'$ and $\Pi_1 \rightarrow \Pi_3\theta$ of this rule imply that the entailment $\Sigma_1 \wedge \Pi_1 \vdash \Sigma_3\theta * \Sigma' \wedge \Pi_3\theta \wedge \Pi_1$ is valid. On the other hand, by applying the entailment substitution law (Theorem 3.1), the hypothesis $H \triangleq \Sigma_3 \wedge \Pi_3 \vdash F_4$ implies that the entailment $\Sigma_3\theta \wedge \Pi_3\theta \vdash F_4\theta$ is valid. It follows that the entailment $\Sigma_3\theta * \Sigma' \wedge \Pi_3\theta \wedge \Pi_1 \vdash F_4\theta * \Sigma' \wedge \Pi_1$ is also valid. We have shown that the two entailments $\Sigma_1 \wedge \Pi_1 \vdash \Sigma_3\theta * \Sigma' \wedge \Pi_3\theta \wedge \Pi_1$ and $\Sigma_3\theta * \Sigma' \wedge \Pi_3\theta \wedge \Pi_1 \vdash F_4\theta * \Sigma' \wedge \Pi_1$ are valid. In addition, this rule's premise gives that $F_4\theta * \Sigma' \wedge \Pi_1 \vdash F_2$ is valid. It follows that the entailment $\Sigma_1 \wedge \Pi_1 \vdash F_2$ in the rule's conclusion is also valid.

Note that in the above argument, we only prove the local soundness of the rule IH. This proof does not require the side conditions about the status of the applied hypothesis ($status = \checkmark$) and the removal of singleton heap predicates ($\exists\, \iota, u, \vec{v}, \Sigma''.(\Sigma' \cong u \overset{\iota}{\mapsto} \vec{v} * \Sigma'')$ and $\exists\, \rho_1, \rho_2.(\rho = \rho_1 @ [(* \mapsto)] @ \rho_2 \wedge (\mathsf{ID}) \notin \rho_1 \wedge (\mathsf{ID}) \in \rho_2)$). Instead, these side conditions will be used to prove the soundness of the entire proof system in the following section.

The soundness of the proof search procedure:

Suppose that when proving an entailment E, the proof search procedure derives a proof tree $\mathcal{T}$ of E. There will be two cases as follows. If the rule IH (*induction hypothesis application*) is not used in $\mathcal{T}$, then induction proof is not used in the proof of E. Since all inference rules are proven to be sound and the proof of E does not require any induction hypotheses, it is clear that the entailment E is valid. If the rule IH is used in $\mathcal{T}$, then the statuses of the applied hypotheses, at the moment they are used, can be either *valid* (✓) or *unknown* (**?**). If the statuses of all the applied hypotheses in the proof tree $\mathcal{T}$ are valid (✓), then it is clear that E is valid, due to the soundness of our inference rules. If there exist some applied hypotheses whose statuses are unknown (**?**), then these hypotheses may participate in a mutual induction proof. We will show that the entailment E is also valid.

Recall that the hypotheses applied by the rule IH are either (i) induction hypotheses recorded by the rule ID, or (ii) hypotheses derived by other rules during proof search. Therefore, in the proof tree $\mathcal{T}$, both the induction hypotheses and other hypotheses can participate in a mutual induction proof. We will transform the proof tree $\mathcal{T}$ into a new tree $\mathcal{T}'$ in which the mutual induction proof involves only the induction hypotheses recorded by the rule ID. This can be done by modifying the rule IH to put the used hypotheses, which are not recorded by the rule ID, into premises of the rule. In particular, suppose the (simplified) rule IH which applies a hypothesis $\Sigma_3 \wedge \Pi_3 \vdash F_4$ (not recorded by the rule ID) into proving a goal entailment $\Sigma_1 \wedge \Pi_1 \vdash F_2$ as follow:

$$\mathsf{IH}\ \frac{F_4\theta * \Sigma' \wedge \Pi_1 \vdash F_2}{\Sigma_1 \wedge \Pi_1 \vdash F_2}\ \begin{array}{l}\text{apply hypothesis: } \Sigma_3 \wedge \Pi_3 \vdash F_4 \text{ (which is not recorded by rule ID)}\\ \exists\, \theta, \Sigma'.(\Sigma_1 \equiv \Sigma_3\theta * \Sigma' \ \wedge\ \Pi_1 \rightarrow \Pi_3\theta)\end{array}$$

We modify the rule IH so that the hypothesis $\Sigma_3 \wedge \Pi_3 \vdash F_4$ appears in the premises as follows:

$$\text{modified } \mathsf{IH}\ \frac{\Sigma_3 \wedge \Pi_3 \vdash F_4 \qquad F_4\theta * \Sigma' \wedge \Pi_1 \vdash F_2}{\Sigma_1 \wedge \Pi_1 \vdash F_2}\ \begin{array}{l}\Sigma_3 \wedge \Pi_3 \vdash F_4 \text{ is not recorded by rule ID}\\ \exists\, \theta, \Sigma'.(\Sigma_1 \equiv \Sigma_3\theta * \Sigma' \ \wedge\ \Pi_1 \rightarrow \Pi_3\theta)\end{array}$$

By modifying the rule IH, we can transform the proof tree $\mathcal{T}$ into a new proof tree $\mathcal{T}'$ (Fig. 11) where every node of $\mathcal{T}$ which applies a hypothesis (not recorded by the rule ID) to prove an entailment is replaced by a new node in $\mathcal{T}'$ that contains not only the target entailment but also full proof tree of the applied hypothesis. Since this transformation is performed only on hypotheses not recorded by the rule ID, it follows that in the new proof tree $\mathcal{T}'$, only induction hypotheses recorded by the rule ID participate in the mutual induction proof.

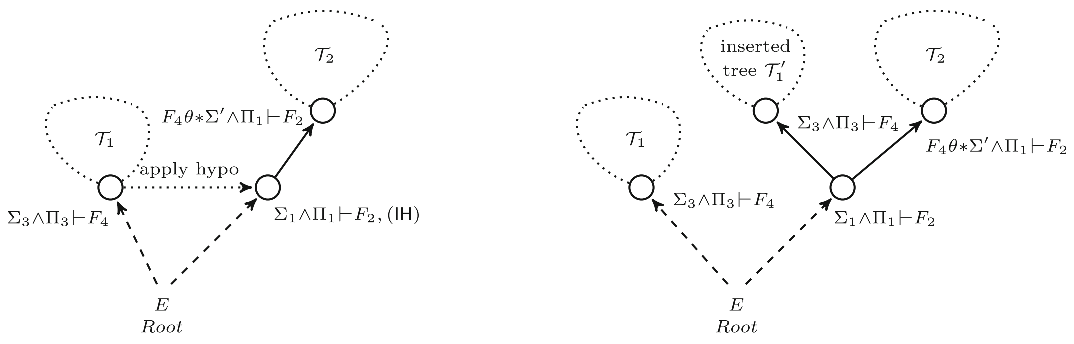
$\mathcal{T}_2$
$F_4\theta{*}\Sigma'{\wedge}\Pi_1{\vdash}F_2$
$\mathcal{T}_1$
apply hypo
$\Sigma_3{\wedge}\Pi_3{\vdash}F_4$
$\Sigma_1{\wedge}\Pi_1{\vdash}F_2$, (IH)
E
Root
inserted tree $\mathcal{T}_1'$
$\mathcal{T}_2$
$\Sigma_3{\wedge}\Pi_3{\vdash}F_4$
$F_4\theta{*}\Sigma'{\wedge}\Pi_1{\vdash}F_2$
$\mathcal{T}_1$
$\Sigma_3{\wedge}\Pi_3{\vdash}F_4$
$\Sigma_1{\wedge}\Pi_1{\vdash}F_2$
E
Root

Fig. 11. Original proof tree $\mathcal{T}$ (left tree), and the transformed proof tree $\mathcal{T}'$ (right tree)

We have shown earlier that all inference rules are proven to be sound. Now, we will show that the new proof tree $\mathcal{T}'$ reflects correctly the mutual induction principle (Theorem 4.3). Let $E_1, \ldots, E_n$ be all induction hypotheses recorded by the rule ID and utilized by the rule IH in the new proof tree $\mathcal{T}'$. Consider an arbitrary separation logic model s, h, for each entailment E_i, with $1 \leq i \leq n$, we will show that: $(\forall\, s', h'.s', h' \prec s, h \rightarrow s', h' \models E_1, \ldots, E_n) \rightarrow s, h \models E_i$.

Indeed, let E_i be $F_i \vdash G_i$. We consider two cases as follows. If $s, h \not\models F_i$, then it is clear that $s, h \models (F_i \vdash G_i)$, by Definition 3.2. Therefore, $s, h \models E_i$.

If $s, h \models F_i$, let $(E_1' \triangleq F_1' \vdash G_1'), \ldots, (E_m' \triangleq F_m' \vdash G_m')$ be all the entailments in the proof tree of E_i, in which the rule IH are applied. Since all inference rules in our proof system do not add new heap predicates when (backward) deriving entailments from their conclusions to their premises, then the model s, h of F_i will be transformed into the models $(s_1', h_1'), \ldots, (s_m', h_m')$ of the antecedents $F_1', \ldots, F_m'$, such that $\mid h_j' \mid \leq \mid h \mid$, with $j = 1, \ldots, m$. When the rule IH applies the induction hypothesis E_a in $E_1, \ldots, E_n$ $(1 \leq a \leq n)$ to prove the entailment E_b' in $E_1', \ldots, E_m'$ $(1 \leq b \leq m)$, the model (s_a'', h_a'') of E_a, which corresponds to s_b', h_b' of E_b', satisfies $\mid h_a'' \mid \leq \mid h_b' \mid$, due to the side condition $\Sigma_1 \cong \Sigma_3\theta * \Sigma'$ in the matching step of the rule IH. If there is a singleton heap which is leftover in this matching step (the side condition $\exists\, \iota, u, \vec{v}, \Sigma''.(\Sigma' \cong u \overset{\iota}{\mapsto} \vec{v} * \Sigma'')$), then the size of the leftover heap is at least 1. Therefore, $\mid h_a'' \mid < \mid h_b' \mid$. Since $\mid h_j' \mid \leq \mid h \mid$, for $j = 1, \ldots, m$, and $1 \leq b \leq m$ then $\mid h_b' \mid \leq \mid h \mid$. It follows that $\mid h_a'' \mid < \mid h \mid$. If there is a removal step of a singleton heap predicate in the proof derivation from E_i to E_b' (the side condition $\exists\, \rho_1, \rho_2.(\rho = \rho_1 @ [(*\mapsto)] @ \rho_2 \wedge (\mathsf{ID}) \notin \rho_1 \wedge (\mathsf{ID}) \in \rho_2)$), then the size of this removed heap predicate is 1. Therefore, $\mid h_b' \mid < \mid h \mid$. Since $\mid h_a'' \mid \leq \mid h_b' \mid$, it follows that $\mid h_a'' \mid < \mid h \mid$. In summary of both these two cases, the side conditions of the rule IH guarantee that when applying the induction hypothesis E_a to prove the derived entailment E_b' of E_i, the model s_a'', h_a'' of E_a, which corresponds to s, h, satisfies that $\mid h_a'' \mid < \mid h \mid$. Therefore, whenever an induction hypothesis E' in $E_1, \ldots, E_n$ is applied to prove a derived entailment of E_i, the model s', h' of that induction hypothesis, which corresponds to s, h, always satisfies that $\mid h' \mid < \mid h \mid$. Since the rule IH has been proven to be sound earlier and all leaves in the proof tree of E_i are empty (derived by axiom rules), this result also implies that the statement $(\forall\, s', h'.s', h' \prec s, h \rightarrow s', h' \models E_1, \ldots, E_n) \rightarrow s, h \models E_i$ holds.

Our above argument is also true for all entailment $E_1, \ldots, E_n$. Hence, the following statement also holds: $(\forall\, s', h'.s', h' \prec s, h \rightarrow s', h' \models E_1, \ldots, E_n) \rightarrow s, h \models E_1, \ldots, E_n$. Since this statement is also the premise of our mutual induction principle (Theorem 4.3), it follows from this principle that all the entailments $E_1, \ldots, E_n$ are valid. Now, we consider two cases regarding the original entailment E. If E is one of the induction hypotheses $E_1, \ldots, E_n$, then E is clearly valid. If E is not one of them, since all inference rules are sound and all the utilized induction hypotheses $E_1, \ldots, E_n$ in the proof tree $\mathcal{T}'$ are valid, then E is evidently valid.

6. Implementation

We have implemented the mutual induction proof system in a prototype prover named Songbird, using the OCaml programming language. In this section, we will focus on discussing the heuristics we apply to make the proof search in Songbird more efficient. These details have not been addressed in the description of the general proof search procedure Prove in the previous section. Our prover Songbird is available online for download at: https://songbird-prover.github.io/mutual-induction/.

We recall that for a given goal entailment, the procedure Prove first finds all applicable inference rules whose conclusions can unify with the goal entailment. Thereafter, it sequentially applies each of the discovered rules until either one of them succeeds to prove the entailment (Prove returns `VALID`), or all of them fail (Prove returns `UNKN`). In our actual implementation, we aim to minimize the effort spent on these tasks. Firstly, we design a heuristic to determine only the necessary inference rules, but not all the possible rules. Secondly, we sort all discovered inference rules by their priorities, i.e., the chance of successfully proving the goal entailment when applying these rules, so that the proof search procedure can selectively apply the rules with higher priorities first. We will elaborate these two details as follows.

To minimize the effort spent on finding inference rules, we first rank all the inference rules by their importance to decide which one should be explored first. We observe that the axiom rules ($\vdash_\pi$, $\bot^\pi_{\mathrm{L}}$ or $\bot^\sigma_{\mathrm{L}}$) are the most important rules, since they do not introduce any new entailments and directly conclude about their goal entailments' validity. Therefore, once the proof search procedure is able to find an axiom rule, it can immediately apply the rule without the need to search for other rules. These rules, thus, will always be explored first by our proof system. The normalization rules ($=_{\mathrm{L}}$, $\exists_{\mathrm{L}}$, $\exists_{\mathrm{R}}$, E_{L} or E_{R}) are the next important rules since they remove surplus constraints from their goal entailments and make them more concise and uniform. This normalization enables the proof search procedure to explore other future inference rules more easily. Therefore, if the proof search procedure is able to find a normalization rule, it also needs to apply this rule early.

We rank the remaining rules ($*\!\mapsto$, $*\mathsf{P}$, P_{R}, ID, and IH) as the last important rules since they all change the heap structures of their goal entailments. In general, we consider these rules to be equally important. Hence, for a given entailment, if the proof search procedure Prove is not able to utilize any axiom rule or normalization rule, it will find all possible instances of the rules $*\!\mapsto$, $*\mathsf{P}$, P_{R}, ID, and IH. Then, depending on the effect that each of the discovered rules produces to the goal entailment, we rank them against the others. We consider the following typical scenarios:

- If Prove discovers an instance of the rule $*\!\mapsto$ which can remove two singleton heap predicates $x{\mapsto}\vec{y}$ and $x{\mapsto}\vec{z}$, each from the two sides of the entailment, then this rule is the most important in comparison with other discovered rules. This is because the two singleton heaps have the same memory address x, and are likely to be identical.
- If Prove identifies an instance of the rule $*\mathsf{P}$ from two inductive heap predicates $\mathsf{P}(\vec{x}_1, y, \vec{z}_1)$ and $\mathsf{P}(\vec{x}_2, y, \vec{z}_2)$, where $|\ \vec{x}_1\ |=|\ \vec{x}_2\ |$ and $|\ \vec{z}_1\ |=|\ \vec{z}_2\ |$, then this rule is the second most important since these two predicates have the same argument y at the same position, hence, they can be identical to be removed.
- If Prove finds an instance of the rules P_{R}, ID or IH which can be applied to produce singleton heap predicates having the same memory address or inductive heap predicates having at least one similar argument, then this rule is also important since it may introduce potentially identical heap predicates, so that the two rules $*\!\mapsto$ or $*\mathsf{P}$ can be applied in the future.

The above observations lead us to design the FindRules procedure to discover applicable inference rules for a given goal entailment. This procedure FindRules is invoked by the proof search procedure Prove, as discussed in Sect. 5.3. In Fig. 12, we provide the pseudocode of FindRules, which implements the aforementioned heuristics. Given an input entailment $F \vdash G$, this procedure sequentially checks if an axiom rule or a normalization rule can be applied for the goal entailment, via the invocation of an auxiliary procedure CanUnifyRule (lines 18–19). If such rule exists, it immediately returns the unified instance of that rule, via the application of another auxiliary procedure CreateRuleInstance (lines 20–21). If FindRules is not able to discover any axiom or normalization rules, it will find all other applicable inference rules among the remaining rules (lines 22–26). These discovered rules will be accumulated into an *ordered set* $\mathcal{R}$ as shown in line 26.

```
Procedure FindRules($\mathcal{H}, \rho, F \vdash G$)
Input: $\mathcal{H}, \rho$ and $F \vdash G$ are respectively a set of hypotheses, a proof trace and a goal entailment.
Output: An ordered set of possible inference rules
18: for each R in { $\vdash_\pi$, $\perp^\pi_L$, $\perp^\sigma_L$, $=_L$, $\exists_L$, $\exists_R$, $E_L$, $E_R$ } do     // Find axiom and normalization rules,
19:     if CanUnifyRule(($\mathcal{H}, \rho, F \vdash G$), R) then
20:         R ← CreateRuleInstance(($\mathcal{H}, \rho, F \vdash G$), R)
21:         return {R}                                                   // and return the first one
22: $\mathcal{R} \leftarrow \varnothing$                                                  // $\mathcal{R}$ is an ordered set
23: for each R in { $*\!\mapsto$, $*$P, $P_R$, ID, IH } do                          // Find all other remaining rules,
24:     if CanUnifyRule(($\mathcal{H}, \rho, F \vdash G$), R) then
25:         R ← CreateRuleInstance(($\mathcal{H}, \rho, F \vdash G$), R)
26:         $\mathcal{R} \leftarrow \mathcal{R} \cup \{R\}$
27: $\mathcal{R}$ ← ReorderRules($\mathcal{R}$, CompareRule)                          // reorder them by priority,
28: return $\mathcal{R}$                                                      // and return all
```

Fig. 12. The rule finding procedure

```
Procedure CompareRule($R_1, R_2$)
Input: $R_1$ and $R_2$ are two instances of the following rules $*\!\mapsto$, $*$P, $P_R$, ID, IH
Output: Priority of the rule $R_1$ in comparison with $R_2$, whose value can be: High, Equal, Low
29: if $R_1 \in \{ *\!\mapsto, *P \}$ then                                       // Compare $*\!\mapsto, *P$ with other rules
30:     if MatchPotentiallyIdenticalPred($R_1$) then return High
31:     if $R_2 \in \{ *\!\mapsto, *P \}$ and MatchPotentiallyIdenticalPred($R_2$) then return Low
32:     if $R_2 \in \{ P_R, ID, IH \}$ and ProducePotentiallyIdenticalPred($R_2$) then return Low
33: if $R_1 \in \{ P_R, ID, IH \}$ then                                     // Comparing $P_R$, ID, IH with other rules
34:     if $R_2 \in \{ *\!\mapsto, *P \}$ and MatchPotentiallyIdenticalPred($R_2$) then return Low
35:     if ProducePotentiallyIdenticalPred($R_1$) then return High
36:     if $R_2 \in \{ P_R, ID, IH \}$ and ProducePotentiallyIdenticalPred($R_2$) then return Low
37: return Equal
```

Fig. 13. The rule comparing procedure

Thereafter, the procedure FindRules sorts all inference rules in $\mathcal{R}$ in an ascending order of their importance (line 27). It utilizes a pair-wise comparison procedure CompareRule to assist in this reordering. The procedure CompareRule is presented in Fig. 13, where its two inputs R_1 and R_2 are instances of the inference rules $*\!\mapsto$, $*$P, P_R, ID, and IH. Its output is one of the three values High, Low, and Equal, indicating that the rule R_1 is respectively more, less, or equally important in comparison to R_2. We implement the heuristics discussed earlier to decide the importance of a rule. In particular, there are three corresponding cases:

– If either R_1 or R_2 is an instance of the two rules $*\!\mapsto$ or $*$P and they match two potentially identical singleton heap predicates of the same memory address or inductive heap predicates having at least one similar argument, which is checked by the procedure MatchPotentiallyIdenticalPred, then the corresponding rule is more important than the other (lines 30, 31, 34).

– If either R_1 or R_2 is an instance of the rules P_R, ID, or IH and they can produce two potentially identical heap predicates, which is checked by the procedure ProducePotentiallyIdenticalPred, then the corresponding is also more important than the other (lines 32, 35, 36).
– If none of the above conditions satisfy, then CompareRule is unable to decide the priority between the two rules R_1 and R_2. In this case, the procedure CompareRule simply returns `Equal` to indicate that these two rules are equally important (line 37).

7. Experimental result

In the previous sections, we have discussed about the mutual induction proof system and the prototype prover Songbird. To evaluate the effectiveness of our prover, we experiment it on two entailment benchmarks from the separation logic literature as well as a synthetic benchmark. We also compare the experimental result of Songbird with the results of other state-of-the-art separation logic provers. Our evaluation was performed on an Ubuntu 14.04 LTS machine with CPU Intel® E5-2620 (2.4GHz) and 64GB RAM. Details of the experiment are available online at: https://songbird-prover.github.io/mutual-induction/.

The first experiment was conducted on two entailment benchmarks sll_entl and slrd_entl from a separation logic competition named SL-COMP in 2014 [SC16]. We consider only *valid entailments* from these two benchmarks since our prover Songbird cannot disprove an entailment, i.e., it cannot conclude if the entailment is invalid. We compare the performance of Songbird with other participating provers of the SL-COMP 2014. They include Slide [IRV14], Sleek [Chi+12], Spen [Ene+14], and Cyclist [BDP11, BGP12]. However, we are unable to make a direct comparison with the induction-based proof technique presented in [CJT15] as their prover was not available by the time we conducted the experiment.

Entailments in the benchmark sll_entl relate only singly linked lists and they are categorized into three groups: bolognesa, clones, and smallfoot, which are named after their original benchmarks. On the other hand, entailments in the benchmark slrd_entl contain inductive heap predicates modeling more diverse data structures. We categorize this benchmarks based on their predicate types, such as singly linked lists (singly-ll), doubly linked lists (doubly-ll), nested lists (nested-ll), skip lists (skip-list), and trees (tree). In Table 1, we report the number of entailments successfully proved by a prover in each category, with a timeout of 120 seconds for proving an entailment, as well as the total and average time spent by each prover to prove these entailments. To ensure a fair comparison, we calculate only the running time that each prover spends on entailments that it can successfully prove. We do not take account of the time spent on unsuccessful entailments since a prover possibly passes a timeout of 120 seconds when it fails to prove an entailment.

In each category, the total number of entailments is shown in the column **#ent**, and the best results among the examined provers are highlighted in bold. Table 1 shows that Songbird can prove more entailments than all the other tools. Particularly, Songbird can prove 308/323 entailments (95.4%) with an average running time of 0.68 seconds per each proved entailment. The second best prover Cyclist can prove only 242/323 entailments (74.9%), which is less than Songbird 66/323 entailments (20.4%). In fact, the difference between this experimental result of Cyclist and Songbird mostly arises in the category bolognesa related to the singly linked list predicate. More specifically, entailments in this category often contain many heap predicates: they contain from 15 to 33 heap predicates, in comparison with the average of 3 to 6 predicates in the entailments of the other categories. This large number of heap predicates often requires significant efforts of an induction-based prover (like Songbird) to syntactically match the antecedent of an induction hypothesis with that of a goal entailment, or of a cyclic-based prover (like Cyclist) to discover downlink among derived entailments. To cope with this difficulty, our prover Songbird aims to apply the unfolding rules and the matching rules as soon as possible to remove potential identical heap predicates from two sides of an entailment, as described earlier in Sect. 6. Certainly, this strategy helps to reduce the size of the complex entailments in the category bolognesa, which makes the induction hypothesis matching step more efficient.

Table 1. Evaluation on the SL-COMP benchmarks, where participants are Slide (Sld), Sleek (Slk), Spen (Spn), Cyclist (Ccl) and our prover Songbird (Sbd)

Benchmark			Proved entailments					Total proving time (s)					Average time (s)				
	Category	#ent	Sld	Slk	Spn	Ccl	Sbd	Sld	Slk	Spn	Ccl	Sbd	Sld	Slk	Spn	Ccl	Sbd
sll_entl	bolognesa	57	0	0	**57**	0	**57**	–	–	17.4	–	140.3	–	–	0.31	–	2.46
	clones	60	0	**60**	**60**	**60**	**60**	–	1.7	2.2	0.2	3.9	–	0.03	0.04	0.00	0.07
	smallfoot	55	0	51	**55**	**55**	**55**	–	1.5	1.6	6.8	3.5	–	0.03	0.03	0.12	0.06
slrd_entl	singly-ll	64	11	48	3	**63**	**63**	0.3	2.0	0.1	1.0	1.8	0.03	0.04	0.03	0.02	0.03
	doubly-ll	37	13	18	9	**29**	25	20.7	1.2	0.3	68.5	0.8	1.59	0.06	0.03	2.36	0.03
	nested-ll	11	0	5	**11**	7	**11**	–	0.8	0.3	9.3	0.4	–	0.16	0.03	1.33	0.04
	skip-list	13	0	4	**13**	5	**13**	–	0.4	0.9	0.2	1.2	–	0.09	0.07	0.04	0.09
	tree	26	13	14	0	23	**24**	60.5	0.9	–	12.9	56.5	4.66	0.07	–	0.56	2.35
	Total	323	37	200	189	242	**308**	81.5	8.5	22.8	99.0	208.6	2.20	0.04	0.11	0.41	0.68

Table 2. Pair-wise comparison between Songbird and other provers on the SL-COMP benchmarks

Songbird	✓$_{\text{sb}}$ ✗$_{\text{o}}$	✗$_{\text{sb}}$ ✓$_{\text{o}}$	✓$_{\text{sb}}$ ✓$_{\text{o}}$	✗$_{\text{sb}}$ ✗$_{\text{o}}$
Cyclist	71	5	237	10
Spen	104	4	204	11
Sleek	109	1	199	14
Slide	281	10	27	5

In theory, we believe that the prover Cyclist can also prove all entailments in the category bolognesa, since its underlying cyclic proof system is similar to our mutual induction proof system. More specifically, these two proof systems have comparable inference rules dealing with heap predicates, and the discovery of downlinks in the cyclic proof system is alike to the application of induction hypotheses in our mutual induction proof system. However, the fact that Cyclist cannot prove any entailment in this category in the timeout of 120 seconds while our prover Songbird can might be due to the different proof search strategies employed by the two provers. We are not aware in details how the proof search strategy of Cyclist is implemented in practice. It is also not mentioned in related papers [BDP11, BGP12]. However, we believe that our current proof search strategy (Sect. 6), if can be integrated into Cyclist, can improve the performance of this prover against the benchmark category bolognesa.

Regarding the proving time, Cyclist is averagely faster than Songbird when it spends about 0.41 seconds per each proved entailment. The two provers Sleek and Spen are slightly behind Cyclist when they can respectively prove 200/323 (61.9%) and 189/323 (58.5%) entailments. The last prover Slide can prove only 37/323 entailments (11.5%). Table 1 also shows that Songbird outperforms other provers in almost all categories, except for doubly-ll. In the doubly-ll category, we are behind solely Cyclist, the second best prover, by only 4/37 entailments in this category. Songbird cannot prove 12/37 entailments in this doubly-ll category since they contain sophisticated constraints in their heap and pure formulas. Our technique may require more effective generalization to handle these unproved entailments.

In Table 2, we make a detailed comparison among Songbird and other provers. Specifically, the first column (✓$_{\text{sb}}$ ✗$_{\text{o}}$) shows the number of entailments that Songbird can prove valid whereas the others cannot. The second column (✗$_{\text{sb}}$ ✓$_{\text{o}}$) 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 (✓$_{\text{sb}}$ ✓$_{\text{o}}$) or cannot (✗$_{\text{sb}}$ ✗$_{\text{o}}$) prove. There are 71 (resp. 104, 109, and 281) entailments that can be proved by our tool, but *not* by Cyclist (resp. Spen, Sleek, and Spen). Furthermore, Songbird can prove almost all entailments that can be proved by other provers. There is only 1 (resp. 4, 5, and 10) over totally 323 entailments that can be proved by Sleek (resp. Spen, Cyclist, and Slide) but not by Songbird. This result is encouraging, thanks to the proposed mutual induction proof technique.

Table 3. Evaluation on the extended slrd_ind entailment benchmark, with the participants Cyclist (Ccl), our prover Songbird (Sbd) and its variant Songbird$_{SI}$ (Sbd$_{SI}$)

slrd_ind Bench		Proved entailments			Total time (s)			Average time (s)		
Category	#ent	Ccl	Sbd$_{SI}$	Sbd	Ccl	Sbd$_{SI}$	Sbd	Ccl	Sbd$_{SI}$	Sbd
ll/ll2	44	33	30	**44**	37.8	3.3	7.8	1.15	0.11	0.18
ll-even/odd	20	**20**	**20**	**20**	17.3	0.9	1.0	0.87	0.05	0.05
ll-left/right	20	**20**	**20**	**20**	13.6	0.7	0.8	0.68	0.03	0.04
misc	32	31	**32**	**32**	121.2	2.4	2.5	3.91	0.07	0.08
Total	116	104	102	**116**	189.9	7.2	12.1	1.83	0.07	0.10

Secondly, we would like to highlight the efficiency of *mutual induction* in our proof technique via a comparison between Songbird and its variant Songbird$_{SI}$, 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, in our earlier work [Ta+16], we designed an entailment benchmark, namely slrd_ind, whose entailments are more complex than those in the slrd_entl benchmark. For example, the slrd_ind benchmark 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[2]. 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. In this work, we improve and extend further this slrd_ind benchmark with new entailments. [3] Entailments in this benchmark are also categorized by their predicate types, including regular singly linked lists (ll/ll2), linked lists with even or odd length (ll-even/odd) and linked list segments which are left- or right-recursively defined (ll-left/right). In addition, entailments in the misc category involve all aforementioned linked list predicates.

As shown in Table 3, Songbird$_{SI}$ is able to prove 102/116 entailments (87.9%), slightly behind the second best prover Cyclist, which can prove 104/116 entailments (89.7%). We do not list other provers Sleek, Spen and Slide in this experiment since they cannot prove any entailment in the benchmark slrd_ind. On the other hand, Songbird, with full capability of mutual induction, can prove all 116 entailments in slrd_ind (100%). In addition, Songbird spends averagely only 0.10 seconds to prove an entailment. This running time is faster than the second best prover Cyclist, which takes averagely 1.83 seconds to prove each entailment. The running time of Songbird is only slightly slower than its variant Songbird$_{SI}$, which needs averagely 0.07 seconds for each proved entailment (0.03 seconds faster than Songbird). A possible reason for the speedy performance of Songbird$_{SI}$is that it only searches for applications of induction hypotheses recorded in a single proof path but not in an entire proof tree as by Songbird. Consequently, Songbird$_{SI}$spends less time on discovering such induction hypothesis applications. However, this feature also prevents Songbird$_{SI}$from utilizing potential hypotheses derived in other paths of the proof tree. In fact, Songbird$_{SI}$cannot prove 14/116 entailments in the benchmark slrd_ind, whereas Songbird can prove all 116/116 entailments. This result is really encouraging as it shows the usefulness and essentials of our mutual induction proof technique.

8. Related work

There has been considerable research on proving entailments in the literature of separation logic. A popular approach is to *pre-define* inductive heap predicates to model specific types of the linked list and the tree data structures. This approach is utilized by Berdine et al. [BCO04, BCO05], Piskac et al. [PWZ13], Bozga et al. [BIP10], and Perez et al. [PR11, PR13]. The authors of these works provide *specific syntax and semantics* for predicates in advance so that they can derive *efficient techniques* to handle these predicates when proving entailments. Since the invented techniques are tied to certain types of pre-defined predicates, they might not be automatically extended to reason about other inductive heap predicates.

[2] The inductive definition of the predicate lsEven is similar to the ListE's definition in Example 3.2.

[3] The extended slrd_entl benchmark is available at https://songbird-prover.github.io/mutual-induction/.

A more general approach is to consider classes of inductive heap predicates satisfying certain *syntactic or semantic restrictions*, such as predicates with a *bounded tree width property* by Iosif et al. [IRS13, IRV14] or predicates describing variants of the linked list data structure by Enea et al. [Ene+14]. These authors propose to prove entailments by translating separation logic entailments into equivalent formulas in theories of automata or graph. Thereby, they can employ developed proof techniques in automata and graph theories to prove the translated formulas, and conclude about the validity of the original separation logic entailments. Nevertheless, inductive heap predicates in this approach might not be able to represent sophisticated properties of data structures such arithmetic constraints about their size or elements' content. These constraints are not directly supported by the considered external theories of graph and automata.

To resolve the above expressiveness limitation, Nguyen and Chin et al. [Ngu+07, NC08, Chi+12], Qiu et al. [Qiu+13, PQM14], and Enea et al. [ESW15] consider classes of *user-defined inductive heap predicates*, i.e., predicates which can be arbitrarily defined by users of verification and analysis systems. These authors propose to prove entailments by using sequent-based proof systems. In their approaches, inductive heap predicates can be handled by special inference rules that remove identical predicates from two sides of an entailment or unfold a predicate by its definition. In general, the proof systems repeatedly apply these rules to simplify a goal entailment until newly derived entailments do not contain any heap predicates in their antecedents or consequents. However, this derivation might be infinite since inductive predicates can be unfolded unlimitedly. To handle such situation, these works require users to provide supplementing lemmas to assist the proof systems to compose, decompose or reorganize inductive heap predicates without using the unfolding rules. The aforementioned proof systems, therefore, are not fully automated.

The above infinite derivation issue in sequent-based proof systems is addressed by interesting research of Brotherston et al. [BDP11] and Chu et al. [CJT15]. These are also the closest to our work. In the first work [BDP11], the authors propose the *cyclic proof system* which allows proof trees to contain cycles. Similar to ours, this proof system also has inference rules to syntactically deal with heap predicates and pure constraints of entailments. Furthermore, the discovery of a downlink in the cyclic proof system is analogous to the application of an induction hypothesis in our mutual induction proof system. However, unlike our work, induction hypotheses are not explicitly identified in the cyclic proof via applications of induction rules; instead, they are implicitly obtained via the discovery of valid cycles in a proof tree.

In the second work, the authors of [CJT15] apply *structural induction* to reason about the recursive structures of inductive heap predicates. Their proof system and ours share similar set of inference rules to handle heap predicates and pure constraints, record and apply induction hypotheses. However, 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, as opposed to our proof system.

9. Limitation

In this section, we will discuss the limitation of our mutual induction proof system. There are two limitations as follows. Firstly, our mutual induction proof system can prove that an entailment is valid, but it cannot conclude if the entailment is invalid. More general, our proof system is an incomplete system. That is, it cannot always decide whether a given entailment is valid or not. This drawback is also a disadvantage of any proof systems for inductive theories containing arithmetic. More specifically, these inductive theories, even when their syntax is restricted to contain only linear arithmetic, can represent any constraints in Peano arithmetic. For example, the multiplication constraint $x \cdot y = z$ can be presented by an inductive predicate $mult(x, y, z)$, which is recursively defined using linear arithmetic as below:

$$mult(x, y, z) \quad \overset{\text{def}}{=} \quad (x{=}0 \wedge z{=}\,0) \quad \vee \quad (x{>}0 \wedge mult(x{-}1, y, z{-}y)) \quad \vee \quad (x{<}0 \wedge mult(x{+}1, y, z{+}y))$$

According to Gödel's second incompleteness theorem [God92], every consistent axiomatic system which includes Peano arithmetic cannot prove its own consistency. Consequently, there does not exist any proof system which can either prove or disprove the validity of any arbitrary formula or entailment containing inductive predicates and linear arithmetic. The incompleteness issue is also carefully addressed in the work of Alan Bundy [Bun01]. Interested readers are invited to refer to it for more details.

Secondly, our proof system currently does not perform necessary generalization of induction hypotheses. We recall that when proving an entailment, the proof system directly records it as an induction hypothesis and unfolds an appropriate inductive heap predicate to derive new entailments. Nevertheless, this induction hypothesis might be too sophisticated to be efficiently handled by an inductive prover. For example, it might contain numerous inductive heap predicates, or complex pure constraints, which overwhelm the inductive hypothesis application step. In fact, such sophisticated entailments appear in the benchmark slrd_entl of SL-COMP 2014. For example, each entailment in the test cases dll-spaghetti.smt2, dll-spaghetti-existential.smt2, dll2-spaghetti.smt2, and dll2-spaghetti-existential.smt2 in this benchmark contains 42 inductive heap predicates of the two predicate symbols DLL_plus and DLL_plus_rev which model doubly linked list data structures. These entailments currently cannot be proved by either our prover Songbird or the prover Cyclist within the timeout of 120 seconds. On the other hand, we believe that Songbird would be able to prove such sophisticated entailments, if it could discover more general entailments, such as $\mathsf{DLL\_plus}(x, y, z, t) \vdash \mathsf{DLL\_plus\_rev}(x, y, z, t)$ or $\mathsf{DLL\_plus\_rev}(x, y, z, t) \vdash \mathsf{DLL\_plus}(x, y, z, t)$. In essence, these new entailments show that the two inductive heap predicates $\mathsf{DLL\_plus}(x, y, z, t)$ and $\mathsf{DLL\_plus\_rev}(x, y, z, t)$ are semantically equivalent. Hence, they can be used as supporting lemmas to transform the original entailments into new entailments containing only one kind of heap predicate, which can be easily proved by both Songbird or Cyclist. Certainly, the discovery of the supporting lemmas is desirable. We have also independently investigated and proposed a framework to automatically synthesize such supporting lemmas in a recent work [Ta+18]. In the future, we wish to integrate this lemma synthesis framework into our proposed mutual induction proof system to strengthen the capability of the latter system in proving entailments.

10. Conclusion

We have proposed a mutual induction principle and developed a proof system to automatically prove separation logic entailments. In particular, we have shown that mathematical induction can perform well on the size of the heap model of separation logic entailments. Our approach allows a goal entailment and other entailments derived during the proof search to be used as hypotheses to mutually prove each other. This mutual hypothesis application theoretically leads to the discovery of more succinct proof trees. We also demonstrate the effectiveness of our induction proof by experimenting on three entailment benchmarks, in which our prototype prover Songbird outperforms the state-of-the-art separation logic provers.

In the future, we would like to empower our proof system with the ability to generalize induction hypotheses as discussed earlier in Sect. 9. We believe that if this feature is well-developed, it will improve the completeness of our proof system and help Songbird to prove more entailments in the slrd_entl benchmark. Lastly, we also wish to integrate our proof system into a practical verification system. We believe that this integration will open up more opportunities to automatically find memory bugs and verify the memory safety of computer programs at large scale.

Acknowledgements

We would like to thank the anonymous reviewers of the Formal Aspect of Computing journal for the careful reading and the constructive comments on our work. The first author wishes to thank Dr. Alwen Fernanto Tiu and an anonymous reviewer of LICS 2018 for the suggestion of using the term *matching*, instead of the term *unification*, when describing the induction hypothesis application. This research is partially supported by an NUS research Grant R-252-000-553-112 and an MoE Tier-2 Grant MOE2013-T2-2-146.

References

[BCI11] Berdine J, Cook B, Ishtiaq S (2011) SLAyer: memory safety for systems-level code. In: International conference on computer aided verification (CAV), pp 178–183

[BCO04] Berdine J, Calcagno C, O'Hearn PW (2004) A decidable fragment of separation logic. In: International conference on foundations of software technology and theoretical computer science (FSTTCS), pp 97–109

[BCO05] Berdine J, Calcagno C, O'Hearn PW (2005) Symbolic execution with separation logic. In: Asian symposium on programming languages and systems (APLAS), pp 52–68

[BDP11] Brotherston J, Distefano D, Petersen RL (2011) Automated cyclic entailment proofs in separation logic. In: International conference on automated deduction (CADE), pp 131–146

[BGP12] Brotherston J, Gorogiannis N, Petersen RL (2012) A generic cyclic theorem prover. In: Asian symposium on programming languages and systems (APLAS), pp 350–367

[BIP10] Bozga M, Iosif R, Perarnau S (2010) Quantitative separation logic and programs with lists. J Autom Reason 45(2):131–156

[Bro+16] Brotherston J, Gorogiannis N, Kanovich MI, Rowe R (2016) Model checking for Symbolic-Heap Separation Logic with inductive predicates. In: Symposium on principles of programming languages (POPL), pp 84–96

[Bro07] Brotherston J (2007) Formalised inductive reasoning in the logic of bunched implications. In: International static analysis symposium (SAS), pp 87–103

[Bun01] Bundy A (2001) The automation of proof by mathematical induction. In: Robinson JA, Voronkov A (eds) Handbook of automated reasoning, vol 2. Elsevier, MIT Press, pp 845–911

[Cal+15] Calcagno C, Distefano D, Dubreil J, Gabi D, Hooimeijer P, Luca M, O'Hearn PW, Papakonstantinou I, Purbrick J, Rodriguez D (2015) Moving fast with software verification. In: NASA international symposium on formal methods (NFM), pp 3–11

[Chi+12] Chin W-N, David C, Nguyen HH, Qin S (2012) Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci Comput Program 77(9):1006–1036

[CJT15] Chu D-H, Jaffar J, Trinh M-T (2015) Automatic induction proofs of data-structures in imperative programs. In: Conference on programming language design and implementation (PLDI), pp 457–466

[Ene+14] Enea C, Lengál O, Sighireanu M, Vojnar T (2014) Compositional entailment checking for a fragment of separation logic. In: Asian symposium on programming languages and systems (APLAS), pp 314–333

[ESW15] Enea C, Sighireanu M, Wu Z (2015) On automated lemma generation for separation logic with inductive definitions. In: International symposium on automated technology for verification and analysis (ATVA), pp 80–96

[God92] Godel K (1992) On formally undecidable propositions of principia mathematica and related systems (Meltzer B, Trans.). Dover Publications, Mineola. ISBN: 0486669807

[Har09] Harrison J (2009) Handbook of practical logic and automated reasoning, 1st edn. Cambridge University Press, New York. ISBN: 0521899575, 9780521899574

[IRS13] Iosif R, Rogalewicz A, Simácek J (2013) The tree width of separation logic with recursive definitions. In: International conference on automated deduction (CADE), pp 21–38

[IRV14] Iosif R, Rogalewicz A, Vojnar T (2014) Deciding entailments in inductive separation logic with tree automata. In: International symposium on automated technology for verification and analysis (ATVA), pp 201–218

[KN87] Kapur D, Narendran P (1987) Matching, unification and complexity. ACM SIGSAM Bull 21(4):6–9

[MB08] De Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: International conference on tools and algorithms for construction and analysis of systems (TACAS), pp 337–340

[NC08] Nguyen HH, Chin W-N (Wei-Ngan) Enhancing program verification with lemmas. In: International conference on computer aided verification (CAV), pp 355–369

[Ngu+07] Nguyen HH, David C, Qin S, Chin W-N (2007) Automated verification of shape and size properties via separation logic. In: International conference on verification, model checking, and abstract interpretation (VMCAI), pp 251–266

[PQM14] Pek E, Qiu X, Madhusudan P (2014) Natural proofs for data structure manipulation in C using separation logic. In: Conference on programming language design and implementation (PLDI), p 46

[PR11] Pérez JAN, Rybalchenko A (2011) Separation Logic + superposition calculus = heap theorem prover. In: Conference on programming language design and implementation (PLDI), pp 556–566

[PR13] Pérez JAN, Rybalchenko A (2013) Separation logic modulo theories. In: Asian symposium on programming languages and systems (APLAS), pp 90–106

[PWZ13] Piskac R, Wies T, Zufferey D (2013) Automating separation logic using SMT. In: International conference on computer aided verification (CAV), pp 773–789

[Qiu+13] Qiu X, Garg P, Stefanescu A, Madhusudan P (2013) Natural proofs for structure, data, and separation. In: Conference on programming language design and implementation (PLDI), pp 231–242

[Rey02] Reynolds JC (2002) Separation logic: a logic for shared mutable data structures. In: Symposium on logic in computer science (LICS), pp 55–74

[Rey08] Reynolds JC (2008) An introduction to separation logic. In: Lecture notes for the PhD fall school on logics and semantics of state, Copenhagen. Retrieved on 2017, March 16th, 2008. http://www.cs.cmu.edu/~jcr/copenhagen08.pdf

[SC16] Sighireanu M, Cok DR (2016) Report on SL-COMP 2014. J Satisf Boolean Model Comput 9:173–186

[Ta+16] Ta Q-T, Le TC, Khoo S-C, Chin W-N (2016) Automated mutual explicit induction proof in separation logic. In: FM 2016: Formal methods—21st international symposium, Limassol, Cyprus, 9–11 Nov 2016, Proceedings. pp 659–676

[Ta+18] Ta Q-T, Le TC, Khoo S-C, Chin W-N (2018) Automated lemma synthesis in symbolic-heap separation logic. In: Symposium on principles of programming languages (POPL), pp 9:1–9:29

Received 10 September 2017
Accepted in revised form 5 September 2018 by Connie Heitmeyer,Ana Cavalcanti, John Fitzgerald, and Stefania Gnesi
Published online 11 October 2018