Completeness of Cyclic Proofs for Symbolic Heaps with Inductive Definitions

Makoto Tatsuta[1], Koji Nakazawa[2], and Daisuke Kimura[3]

[1] National Institute of Informatics / Sokendai, Japan, `tatsuta@nii.ac.jp`
[2] Nagoya University, Japan, `knak@is.nagoya-u.ac.jp`
[3] Toho University, Japan, `kmr@is.sci.toho-u.ac.jp`

Abstract. Separation logic is successful for software verification in both theory and practice. Decision procedure for symbolic heaps is one of the key issues. This paper proposes a cyclic proof system for symbolic heaps with general form of inductive definitions called cone inductive definitions, and shows its soundness and completeness. Cone inductive definitions are obtained from bounded-treewidth inductive definitions by imposing some restrictions for existentials, but they still include a wide class of recursive data structures. The completeness is proved by using a proof search algorithm and it also gives us a decision procedure for entailments of symbolic heaps with cone inductive definitions. The time complexity of the algorithm is nondeterministic double exponential. A prototype system for the algorithm has been implemented and experimental results are also presented.

1 Introduction

Separation logic is successful for software verification [6, 7, 25]. Several systems based on this idea have been actively investigated and implemented. One of the keys of these systems is the entailment checker that decides the validity of a given entailment of symbolic heaps, which are restricted forms of separation logic formulas.

Inductive predicates are used to describe recursive data structures such as lists and trees. In order to verify programs with recursive data structures, symbolic heaps with inductive predicates are useful. Our purpose is to obtain a decision procedure for entailments of symbolic heaps with general inductive definitions.

The validity of entailments for symbolic heaps with inductive definitions is known to be undecidable [1]. Hence it is important to find an expressive class of inductive definitions such that we have efficient decision procedure. Iosif et al. [19] proposed the system SLRD$_{btw}$, which is the first decidable system for entailments of symbolic heaps with general inductive definitions. The condition on inductive definitions imposed in [19] is called the *bounded-treewidth condition*, which is one of the most flexible conditions for decidability.

In [19], the decision procedure is given by model theoretic engine. On the other hand, proof theoretic engine, namely proof search based on some proof theory, has several advantages. (1) We can extend it by adding new inference rules. (2) It provides reasons for the output. (3) It may be fast when the input entailment is valid. (4) We can easily change it into a semi-decision procedure by adding some heuristics for proof search. There are some decision procedure based on proof systems for separation logic [6, 7]. However, they have only hard-coded predicates such as only lists and trees. Therefore, it is important to find an expressive class of inductive definitions such that we can present decision procedure based on a proof system.

Many proof systems with inductive definitions use Martin-Löf style introduction and elimination rules to describe inductive predicates. On the other hand, cyclic-proof systems [10, 11] may give us a better system for our purpose. In cyclic proof systems, the induction is represented as a cyclic structure, where some open assumptions are allowed as induction hypotheses. This mechanism gives us more efficient proof search, since we need not fix an induction formula in advance. In fact, some systems (explicitly or implicitly) based on cyclic-proof systems have been proposed for the entailment checking problem for separation logic [10, 11, 15, 29, 30]. However it is not well studied on complete cyclic-proof systems for separation logic.

Our contributions are: (1) We propose a new class of inductive definitions, called the *cone inductive definitions*. This class is useful since it is sufficiently large, and we can present a proof system with the following nice properties. (2) We propose a new cyclic proof system for entailment with symbolic heaps and cone inductive definitions. (3) We will show that our proof system is complete. (4) We present a proof search algorithm for the proof system, which decides the validity of entailment. (5) In order to make these possible, we develop new ideas: (a) a new rule for *spatial factorization*, called the *factor rule*, to divide a spatial atomic formula, (b) a new rule for the split rule that can handle disjunction in the succedent, and (c) an auxiliary extension of the language for explicit case analysis. Furthermore, in order to confirm that our results are useful, we implemented a prototype system based on our proof-search algorithm and present some experimental results.

The class of the *cone inductive definitions* is obtained from the class of the bounded-treewidth by imposing an additional condition that the definition body itself guarantees existentials to be allocated. The cone inductive definition requires that every definition clause of the inductive predicate $P(x, \overrightarrow{y})$ contains $x \mapsto (t)$ for some t. We call the first argument x a *root* of $P(x, \overrightarrow{y})$, since it is the root of a tree-like structure described by the predicate. An example of cone inductive definitions is the following singly-linked list: $\mathrm{ls}(x, y) =_{\mathrm{def}} x \mapsto y \vee \exists z(x \mapsto (z) * \mathrm{ls}(z, y))$, where it is a bounded tree-width inductive definition and the existential z is guaranteed to be allocated by $\mathrm{ls}(z, y)$ in the definition clause. The cone inductive definitions are still quite expressive, since this class contains doubly-linked lists, skip lists, nested lists, and so on.

Based on the Unfold-Match strategy to the cone inductive definitions and the cyclic proofs, our algorithm will use the following *Unfold-Match-Remove-Split* strategy (written as U-M-R-S). First, unfold predicates of the same root x

in both antecedent and succedent. Next, the existentials of the antecedent are replaced by fresh variables and instantiate the existentials of the succedent by matching. Then, remove the same $x \mapsto (t)$ in both antecedent and succedent. Finally we split $F_1 * F_2 \vdash G_1 * G_2$ into $F_1 \vdash G_1$ and $F_2 \vdash G_2$ by the split rule of separation logic. We repeat these steps for each subgoal.

However the U-M-R-S strategy may get stuck when we cannot find predicates which have a common root on both sides. In order to solve this problem, we propose a new inference rule called the factor rule, which uses *inductive wands*. The inductive wand $P \mathrel{-\!\!*}^{\text{ind}} Q$ represents a subheap of Q which is obtained by removing a heap which satisfies P. The inductive wands are also predicates defined by cone inductive definitions. By the inductive wand, we can divide a predicate $P(x)$ in into two predicates $Q(y) \mathrel{-\!\!*}^{\text{ind}} P(x)$ and $Q(y)$, where we can find the root y in $Q(y)$ which occurs as a root on both sides. The definition clauses of the inductive wand $Q(y) \mathrel{-\!\!*}^{\text{ind}} P(x)$ are automatically generated from the definitions of $P(x)$ and $Q(y)$, and they satisfy the condition of the cone inductive definition. We will also use the factor rule in order to make the allocated variables of the antecedent and each disjunct of the succedent coincide when we apply the split rule.

A similar idea to the inductive wand has been proposed as the inductive segment in [5], where they used it for software verification by abstract interpretation. By the inductive segment, they can unfold inductive predicates in the reverse direction to find some fixed points in shape analysis. On the other hand, we will use inductive wands for a different purpose, namely, deciding entailments of symbolic heaps.

We will show the class of cone inductive predicates is closed under inductive wands. It is not easy to obtain a useful class of inductive definition that is closed under inductive wands. For example, the class of inductive definitions with bounded treewidth is not closed under inductive wands. Magic wands often causes undecidability [4]. By this closure property, it is not necessary to add any special clauses to the definition of semantics for inductive wands.

The split rule is new since it handles disjunctions in the succedent. The ideas for the split rule will be fully explained in Section 4.

For completeness, we extend the succedent to disjunctions and introduce predicates $t \downarrow$ and $t \Uparrow$, where $t \downarrow$ means that t is in the domain of the heap, and $t \Uparrow$ means that t is not related to the heap. They will be defined in Section 3 and explained in Section 4.

Many entailment checkers for symbolic heaps with inductive definitions have been discussed. Several of them do not have general inductive definitions and have only hard-coded inductive predicates [6, 7, 16, 22, 23]. The entailment checkers for general inductive definitions are studied in [10, 11, 14, 15, 17–21, 24, 29–31]. The systems in [17–21, 24, 31] are all model theoretic. The algorithm in [21] has better time complexity than ours. The system in [20] does not have disequality. The systems in [10, 11, 15, 29, 30] use cyclic proofs, but neither of them is complete. The system in [14] is based on ordinary sequent calculus and is not complete.

The cyclic proofs have been intensively investigated for the first-order predicate logic [2,3,9,11,26], a bunched implication system [8], and a symbolic heap system [10,11].

Section 2 defines separation logic with inductive definitions. In Section 3 we extend our language. In Section 4, we explain main ideas. Section 5 defines the system CSLID$^{\omega}$ and proves its soundness. In Section 6 we give the proof search algorithm, and show its partial correctness and termination. Section 7 proves the completeness of CSLID$^{\omega}$. Section 8 presents experimental results. We conclude in Section 9.

2 Symbolic Heaps with Inductive Definitions

This section defines symbolic heaps, inductive definitions, and their semantics.

We will use vector notation $\overrightarrow{x}$ to denote a sequence $x_1, \ldots, x_k$ for simplicity. $|\overrightarrow{x}|$ denotes the length of the sequence. Sometimes we will also use a notation of a sequence to denote a set for simplicity. Then we will write $\overrightarrow{p} + \overrightarrow{q}$ for the disjoint union set of the sets $\overrightarrow{p}$ and $\overrightarrow{q}$. We write $\equiv$ for the syntactical equivalence.

2.1 Language

Our language is a first order language with a new connective $*$ and inductive predicates, and defined as follows.

First-order variables $x, y, z, w, v, \ldots$.

Terms $t, u, p, q, r ::= x \mid \mathrm{nil}$. Inductive Predicate Symbols $P, Q, \ldots$.

We define formulas F, G of separation logic as those of the first-order language generated by these terms, the propositional constant emp, predicate symbols $=$, $\mapsto$, P,Q,..., and an additional logical connective $*$. We write $t \neq u$ for $\neg t = u$. We assume some number n_{cell} for the number of elements in a cell. Next we define symbolic heaps.

Pure formulas $\Pi ::= t = t \mid t \neq t \mid \Pi \wedge \Pi$.

Spatial formulas $\Sigma ::= \mathrm{emp} \mid t \mapsto (t_1, \ldots, t_{n_{\mathrm{cell}}}) \mid P(\overrightarrow{t}) \mid \Sigma * \Sigma$.

We suppose $*$ binds more tightly than $\wedge$. We will sometimes write $P(t)$ for $P(t, \overrightarrow{t})$. We write $*_{i \in [1,n]} P_i(x_i)$ for $P_1(x_1) * \ldots * P_n(x_n)$. Similarly we write $*_{i \in I} P_i(x_i)$. We write $\Pi \subseteq \Pi'$ when all the conjuncts of Π are contained in those of Π'.

qf-Symbolic Heaps $A, B ::= \Pi \wedge \Sigma \mid \Sigma$. Symbolic Heaps $\phi ::= \exists \overrightarrow{x} A$.

Entailments $A \vdash B_1, \ldots, B_n$ where the succedent is a set $\{\Phi_1, \ldots, \Phi_n\}$.

The language has inductive definitions of inductive predicates.

Inductive Definitions $P(x, \overrightarrow{y}) =_{\mathrm{def}} \bigvee_i \phi_i(x, \overrightarrow{y})$ where ϕ_i is a definition clause.

Definition Clauses $\phi(x, \overrightarrow{y}) \equiv \exists \overrightarrow{z} (\Pi \wedge x \mapsto (\overrightarrow{u}) * *_{i \in I} P_i(z_i, \overrightarrow{t}_i))$, where

- $\{z_i \mid i \in I\} \subseteq \overrightarrow{u}$ (connectivity).
- $\overrightarrow{z} = \{z_i \mid i \in I\}$ (strong establishment).

We call the first argument x of a spatial atomic formula $P(x, \overrightarrow{t})$ a *root*.

The strong establishment implies the establishment condition required by the bounded-treewidth condition. These conditions guarantee that the cell at address x decides the content of every existential variable. It is similar to the constructively valued condition in [13].

We give some examples of the inductive definitions in the following.

The list segment: $\mathrm{ls}(x, y) =_{\mathrm{def}} x \mapsto y \vee \exists z(x \mapsto (z) * \mathrm{ls}(z, y))$.

The doubly-linked list: $\mathrm{dll}(h, p, n, t) =_{\mathrm{def}} h = t \wedge h \mapsto (p, n) \vee \exists z(h \mapsto (p, z) * \mathrm{dll}(z, h, n, t))$.

The nested list: $\mathrm{listnest}(x) =_{\mathrm{def}} \exists z(x \mapsto (z, \mathrm{nil}) * \mathrm{ls}(z, \mathrm{nil})) \vee \exists z_1 z_2(x \mapsto (z_1, z_2) * \mathrm{ls}(z_1, \mathrm{nil}) * \mathrm{listnest}(z_2))$.

The nested list segment:

$\mathrm{lsnest}(x, y) =_{\mathrm{def}} \exists z(x \mapsto (z, \mathrm{nil}) * \mathrm{ls}(z, y)) \vee \exists z_1 z_2(x \mapsto (z_1, z_2) * \mathrm{ls}(z_1, y) * \mathrm{lsnest}(z_2, y))$.

The skip list: $\mathrm{skl1}(x, y) =_{\mathrm{def}} x \mapsto (\mathrm{nil}, y) \vee \exists z(x \mapsto (\mathrm{nil}, z) * \mathrm{skl1}(z, y))$,

$\mathrm{skl2}(x, y) =_{\mathrm{def}} \exists z(x \mapsto (y, z) * \mathrm{skl1}(z, y)) \vee \exists z_1 z_2(x \mapsto (z_1, z_2) * \mathrm{skl1}(z_2, z_1) * \mathrm{skl2}(z_1, y))$.

Many examples in [10] are definable in our system as follows: List, ListE, ListO are definable, RList is not definable. DLL, PeList, SLL, BSLL, BinTree, BinTreeSeg, BinListFirst, BinListSecond, BinPath are not definable but will be definable in a straightforward extension of our system by handling emp in the base cases.

We prepare some notions. We define $P^{(m)}$ by

$$\begin{aligned} &P^{(0)}(\overrightarrow{x}) \equiv (\mathrm{nil} \neq \mathrm{nil}), \\ &P^{(m+1)}(\overrightarrow{x}) \equiv \bigvee_i \phi_i[P := P^{(m)}], \end{aligned}$$

where $P(\overrightarrow{x}) =_{\mathrm{def}} \bigvee_i \phi_i$. $P^{(m)}$ is m-time unfold of P. We define $F^{(m)}$ as obtained from F by replacing every inductive predicate P by $P^{(m)}$.

We write T for a finite set of terms. We define $(\neq (T_1, T_2))$ as $\bigwedge_{t_1 \in T_1, t_2 \in T_2, t_1 \not\equiv t_2} t_1 \neq t_2$. We write $y \neq \overrightarrow{t}$ for $(\neq (\{y\}, \overrightarrow{t}))$. We define $(\neq (T))$ as $(\neq (T \cup \{\mathrm{nil}\}, T))$.

2.2 Semantics

This subsection gives semantics of the language.

We define the following structure: Vars is the set of variables, $\mathrm{Val} = N$, $\mathrm{Locs} = \{x \in N | x > 0\}$, $\mathrm{Heaps} = \mathrm{Locs} \to_{fin} \mathrm{Val}^{n_{\mathrm{cell}}}$, $\mathrm{Stores} = \mathrm{Vars} \to \mathrm{Val}$. Each $s \in \mathrm{Stores}$ is called a *store*. Each $h \in \mathrm{Heaps}$ is called a *heap*, $\mathrm{Dom}(h)$ is the domain of h, and $\mathrm{Range}(h)$ is the range of h. We write $h = h_1 + h_2$ when $\mathrm{Dom}(h_1)$ and $\mathrm{Dom}(h_2)$ are disjoint and the graph of h is the union of those of h_1 and h_2. A pair (s, h) is called a *heap model*, which means a memory state. The value $s(x)$ means the value of the variable x in the model (s, h). Each value $a \in \mathrm{Dom}(h)$

means an address, and the value of $h(a)$ is the content of the memory cell at address a in the heap h. We suppose each memory cell has $n_{\rm cell}$ elements as its content.

The interpretation $s(t)$ for any term t is defined as 0 for nil and $s(x)$ for the variable x. For a formula F we define the interpretation $s, h \models F$ as follows.

$s, h \models t_1 = t_2$ if $s(t_1) = s(t_2)$,

$s, h \models F_1 \wedge F_2$ if $s, h \models F_1$ and $s, h \models F_2$,

$s, h \models \neg F$ if $s, h \not\models F$,

$s, h \models$ emp if $\mathrm{Dom}(h) = \emptyset$,

$s, h \models t \mapsto (t_1, \ldots, t_{n_{\rm cell}})$ if $\mathrm{Dom}(h) = \{s(t)\}$ and $h(s(t)) = (s(t_1), \ldots, s(t_{n_{\rm cell}}))$,

$s, h \models F_1 * F_2$ if $s, h_1 \models F_1$ and $s, h_2 \models F_2$ for some $h_1 + h_2 = h$,

$s, h \models P(\overrightarrow{t})$ if $s, h \models P^{(m)}(\overrightarrow{t})$ for some m,

$s, h \models \exists z F$ if $s[z := b], h \models F$ for some $b \in \mathrm{Val}$.

We write $A \models B_1, \ldots, B_n$ for $\forall sh(s, h \models A \to ((s, h \models B_1) \vee \ldots \vee (s, h \models B_n)))$. The entailment $A \vdash B_1, \ldots, B_n$ is said to be valid if $A \models B_1, \ldots, B_n$ holds. Our goal in this paper is to decide the validity of a given entailment.

For saving space, we identify some syntactical objects that have the same meaning, namely, we use implicit transformation of formulas by using the following properties: $\wedge$ is commutative, associative, and idempotent; $*$ is commutative, associative, and has the unit emp; $=$ is symmetric; $\exists x G \leftrightarrow G$, $\exists x(F \wedge G) \leftrightarrow \exists x F \wedge G$, and $\exists x(F * G) \leftrightarrow \exists x F * G$, when F, G are formulas and $x \notin \mathrm{FV}(G)$; $\Pi \wedge (F * G) \leftrightarrow (\Pi \wedge F) * G$.

3 Language Extension

3.1 Extended Language

In this section, we extend our language from symbolic heaps by $\downarrow$, $\Uparrow$ and $-\!\!*^{\rm ind}$, since they are necessary to show the completeness.

We extend inductive predicate symbols with $Q_1 -\!\!*^{\rm ind} \ldots -\!\!*^{\rm ind} Q_m -\!\!*^{\rm ind} P$ where $Q_1, \ldots, Q_m, P$ are original inductive predicate symbols. We call m the *depth of wands*. We write $Q_1(\overrightarrow{t}_1) -\!\!*^{\rm ind} \ldots -\!\!*^{\rm ind} Q_m(\overrightarrow{t}_m) -\!\!*^{\rm ind} P(\overrightarrow{t})$ for $(Q_1 -\!\!*^{\rm ind} \ldots -\!\!*^{\rm ind} Q_m -\!\!*^{\rm ind} P)(\overrightarrow{t}, \overrightarrow{t}_1, \ldots, \overrightarrow{t}_m)$.

We extend our first-order language with the extended inductive predicate symbols and unary predicate symbols $\downarrow$ and $\Uparrow$. $t \downarrow$ means that t is in $\mathrm{Dom}(h)$ and $t \Uparrow$ means that t is unrelated to the heap. We write $t \uparrow$ for $\neg t \downarrow$. Then $t \uparrow$ means that t is not in the domain of the heap.

We write F, G, Σ, A, B, ϕ for the same syntactical objects with the extended inductive predicate symbols. We also extend definition clauses with the extended inductive predicate symbols. We use X, Y for a finite set of variables and write $X \uparrow$ for $\bigwedge\{t \uparrow \mid t \in X\}$. $X \downarrow$ and $X \Uparrow$ are similarly defined. We write $\exists \overrightarrow{x} \downarrow$ for $\exists \overrightarrow{x}(\overrightarrow{x} \downarrow \wedge \ldots)$. Similarly we write $\exists \overrightarrow{x} \Uparrow$.

We define: $\mathcal{P} ::= \mapsto \mid P$ where P varies in inductive predicate symbols,

Cones $\Delta ::= \mathcal{P}(\overrightarrow{t}) \wedge X \downarrow$, and $\Gamma ::= \Delta \mid \Gamma * \Gamma$, and $\psi ::= Y \uparrow \wedge \Pi \wedge \Gamma$, and $\Phi ::= \exists \overrightarrow{x} \exists \overrightarrow{y} \Uparrow (\Pi \wedge \Gamma)$.

A cone $\mathcal{P}(\overrightarrow{t}) \wedge X \downarrow$ is a unit of our spatial formula. It means that the atomic formula $\mathcal{P}(\overrightarrow{t})$ specifies the heap as well as the variables X are allocated in the heap. Γ is a separating conjunction of cones. ψ will be used as an antecedent of our extended entailment and has more information that the variables Y are not allocated in the heap. Φ will be used in a succedent of our extended entailment and it existentially quantifies variables $\overrightarrow{x}\overrightarrow{y}$ with information that the variables $\overrightarrow{y}$ are not related to the heap.

We define entailments as $\psi \vdash \Phi_1, \ldots, \Phi_n$.

We write J for an entailment. In ψ, Φ, we call Γ a *spatial part* and Π a *pure part.*

We define $\mathrm{Roots}(X \Uparrow \wedge Y \uparrow \wedge \Pi \wedge *_{i \in I}(\mathcal{P}_i(x_i, \overrightarrow{t}_i) \wedge X_i \downarrow)) = \{x_i | i \in I\}$. Then we define $\mathrm{Roots}(\exists x \Phi) = \mathrm{Roots}(\Phi)$ if $x \notin \mathrm{Roots}(\Phi)$, and undefined otherwise. We define $\mathrm{Cells}(X \Uparrow \wedge Y \uparrow \wedge \Pi \wedge *_{i \in I}(X_i \downarrow \wedge \mathcal{P}_i(x_i))) = \bigcup_{i \in I} X_i$. Then we define $\mathrm{Cells}(\exists x \Phi) = \mathrm{Cells}(\Phi) - \{x\}$. We write $\mathrm{Alloc}(F)$ for $\mathrm{Roots}(F) \cup \mathrm{Cells}(F)$ and call them *allocated variables* of F. $\mathrm{Alloc}(F)$ means the set of variables allocated in the heap.

We define a substitution as a map from the set of variables to the set of terms. For a substitution θ, we define $\mathrm{Dom}(\theta) = \{x | \theta(x) \not\equiv x\}$ and $\mathrm{Range}(\theta) = \{\theta(x) | x \in \mathrm{Dom}(\theta)\}$. We define a *variable renaming* as a substitution that is a bijection among variables with a finite domain.

We define semantics of the extended language.

Definition 3.1 $s, h \models t \downarrow$ if $s(t) \in \mathrm{Dom}(h)$.

$s, h \models t \Uparrow$ if $s(t) \notin \mathrm{Range}(h) \cup \mathrm{Dom}(h)$.

We say $\psi \vdash \Phi_1, \ldots, \Phi_n$ is valid when for all s, h, if $s, h \models \psi$ then there is some i such that $s, h \models \Phi_i$. We write $\psi \models \Phi_1, \ldots, \Phi_n$ when $\psi \vdash \Phi_1, \ldots, \Phi_n$ is valid.

For saving space, we identify some syntactical objects that have the same meaning, namely, we use implicit transformation of formulas by using the following property: $(F * G) \wedge X \Uparrow \leftrightarrow (F \wedge X \Uparrow) * (G \wedge X \Uparrow)$.

3.2 Inductive Wand

This section gives the definition clauses for inductive predicates that contain the inductive wand.

Definition 3.2 The definition clauses of $Q(y, \overrightarrow{w}) —*^{\mathrm{ind}} P(x, \overrightarrow{y})$ are as follows:

Case 1. $\exists(\overrightarrow{z} - z_i)((\overrightarrow{w} = \overrightarrow{t}_i \wedge \Pi \wedge x \mapsto (\overrightarrow{u}) * *_{l \neq i} P_l(z_l, \overrightarrow{t}_l))[z_i := y])$ where $Q = P_i$ and $\exists \overrightarrow{z}(\Pi \wedge x \mapsto (\overrightarrow{u}) * *_l P_l(z_l, \overrightarrow{t}_l))$ is a definition clause of $P(x, \overrightarrow{y})$.

Case 2. $\exists \overrightarrow{z}(\Pi \wedge x \mapsto (\overrightarrow{u}) * *_{l \neq i, l \in L} P_l(z_l, \overrightarrow{t}_l) * (Q(y, \overrightarrow{w}) —*^{\mathrm{ind}} P_i(z_i, \overrightarrow{t}_i)))$ where $i \in L$ and $\exists \overrightarrow{z}(\Pi \wedge x \mapsto (\overrightarrow{u}) * *_{l \in L} P_l(z_l, \overrightarrow{t}_l))$ is a definition clause of $P(x, \overrightarrow{y})$.

$Q(y) —*^{\mathrm{ind}} P(x)$ is inductively defined by the definition clauses obtained by removing $Q(y)$ from the definition clauses of $P(x)$. The inductive wand

$Q(y) \mathrel{—\!\!*}^{\mathrm{ind}} P(x)$ plays a similar role to the ordinary magic wand $Q(y) \mathrel{—\!\!*} P(x)$, but it is stronger than the ordinary magic wand and it is defined syntactically. Roughly speaking, it is defined to be false if it cannot be defined syntactically.

Example 3.3 $\mathrm{ls}(y,v) \mathrel{—\!\!*}^{\mathrm{ind}} \mathrm{ls}(x,w) =_{\mathrm{def}} w = v \wedge x \mapsto (y) \vee \exists z(x \mapsto (z) * (\mathrm{ls}(y,v) \mathrel{—\!\!*}^{\mathrm{ind}} \mathrm{ls}(z,w)))$.

We can show $P(x) \mathrel{—\!\!*}^{\mathrm{ind}} Q(y) \mathrel{—\!\!*}^{\mathrm{ind}} R(z)$ and $Q(y) \mathrel{—\!\!*}^{\mathrm{ind}} P(x) \mathrel{—\!\!*}^{\mathrm{ind}} R(z)$ are equivalent.

Lemma 3.4 *If $\overrightarrow{R}$ is $P_1(\overrightarrow{t}_1), \ldots, P_n(\overrightarrow{t}_n)$ and $\overrightarrow{R}'$ is its permutation, then $\overrightarrow{R}' \mathrel{—\!\!*}^{\mathrm{ind}} P(\overrightarrow{t})$ is equivalent to $\overrightarrow{R} \mathrel{—\!\!*}^{\mathrm{ind}} P(\overrightarrow{t})$.*

We have an elimination rule for inductive wands.

Lemma 3.5 (Strong Wand Elimination) $(Q(y,\overrightarrow{w}) \mathrel{—\!\!*}^{\mathrm{ind}} P(x,\overrightarrow{z})) * (\overrightarrow{R(v,\overrightarrow{u})} \mathrel{—\!\!*}^{\mathrm{ind}} Q(y,\overrightarrow{w})) \models \overrightarrow{R(v,\overrightarrow{u})} \mathrel{—\!\!*}^{\mathrm{ind}} P(x,\overrightarrow{z})$.

We define $\mathrm{Dep}(P)$ as the set of inductive predicate symbols that appear in the unfolding of P.

We have an introduction rule for inductive wands.

Lemma 3.6 (Strong Wand Introduction) $x \neq y \wedge y \downarrow \wedge (\overrightarrow{R(v,\overrightarrow{u})} \mathrel{—\!\!*}^{\mathrm{ind}} P(x,\overrightarrow{z})) \models \{\exists \overrightarrow{w}((Q(y,\overrightarrow{w}), \overrightarrow{R_1(v_1,\overrightarrow{u}_1)} \mathrel{—\!\!*}^{\mathrm{ind}} P(x,\overrightarrow{z})) * (\overrightarrow{R_2(v_2,\overrightarrow{u}_2)} \mathrel{—\!\!*}^{\mathrm{ind}} Q(y,\overrightarrow{w})) \mid \overrightarrow{R(v,\overrightarrow{u})} = (\overrightarrow{R_1(v_1,\overrightarrow{u}_1)} + \overrightarrow{R_2(v_2,\overrightarrow{u}_2)}), Q \in \mathrm{Dep}(P), \overrightarrow{R}_2 \subseteq \mathrm{Dep}(Q)\}$

4 Main Ideas

We explain our main ideas of our contributions. They are for the language extension and the proof search algorithm.

A rule is defined to be *locally complete* if all its assumptions are valid when its conclusion is valid.

(1) We extend our language to $t \downarrow$ and $t \uparrow$ and disjunction in a succedent. We use them to case analysis. One of advantages by this case analysis is to guide how to use the split rule. For a formula F, by case analysis, we can assume every variable x is in $\mathrm{Alloc}(F)$ or has $x \uparrow$ in F. Then we can impose the condition $\mathrm{Alloc}(F_1) = \mathrm{Alloc}(G_1)$ and $\mathrm{Alloc}(F_2) = \mathrm{Alloc}(G_2)$ to use the split rule

$$\frac{F_1 \vdash G_1 \quad F_2 \vdash G_2}{F_1 * F_2 \vdash G_1 * G_2}$$

without loss of generality in proof search, since a valid subgoal has the same allocated variables on both sides.

Disjunction of a succedent is necessary for completeness. If we unfold $P(x)$ in the succedent of $A \vdash P(x) * B$ and $P(x)$ has two definition clauses $R_1(x)$ and

$R_2(x)$, we need $A \vdash ((R_1(x) * B) \vee (R_2(x) * B))$ to preserve the validity of the entailment by unfolding. We will write $A \vdash R_1(x) * B, R_2(x) * B$ for it.

(2) Our new inference rules are the factor rule and the split rule. The factor rule transforms $P(x)$ into $(Q(y) \mathrel{—\!\!*}^{\mathrm{ind}} P(x)) * Q(y)$, where $Q(y) \mathrel{—\!\!*}^{\mathrm{ind}} P(x)$ is an atomic formula with another inductive predicate $Q(-) \mathrel{—\!\!*}^{\mathrm{ind}} P(-)$, which is called an *inductive wand*. The factor rule exposes a hidden y as the root of the inductive predicate Q, and this will be used for the unfolding step in our proof search algorithm. Since the factor rule divides a single atomic formula $P(x)$ into two atomic formulas $Q(y) \mathrel{—\!\!*}^{\mathrm{ind}} P(x)$ and $Q(y)$ where x is allocated in one formula and y is allocated in the other formula, this will also enable us to split separating conjunction according to allocated variables in our proof search algorithm.

We may consider a nested inductive wand like $R(-) \mathrel{—\!\!*}^{\mathrm{ind}} (Q(-) \mathrel{—\!\!*}^{\mathrm{ind}} P(-))$. We will show that the depth of necessary inductive wands for our proof search algorithm has a fixed upper bound.

(3) The split rule is new since it handles disjunctions in the succedent.

For our proof search algorithm, we need the split rule that keeps validity. A naive split rule does not keep validity. For example, consider $F_1 * F_2 \vdash G_1^1 * G_2^1, G_1^2 * G_2^2$ where $\mathrm{Alloc}(F_1) = \mathrm{Alloc}(G_1^i)$ and $\mathrm{Alloc}(F_2) = \mathrm{Alloc}(G_2^i)$ for $i = 1, 2$. Then, if $s, h_1 \models F_1$ and $s, h_2 \models F_2$, and $s, h_1 + h_2 \models G_1^i * G_2^i$, then $s, h_1 \models G_1^i$ and $s, h_2 \models G_2^i$ because of the shape of a heap for a cone. Then we have $F_1 \vdash G_1^1, G_1^2$ and $F_2 \vdash G_2^1, G_2^2$. If we transform it into $F_1 \vdash G_1^1, G_1^2$ and $F_2 \vdash G_2^1, G_2^2$, this transformation does not keep validity, namely, the naive split rule

$$\frac{F_1 \vdash G_1^1, G_1^2 \quad F_2 \vdash G_2^1, G_2^2}{F_1 * F_2 \vdash G_1^1 * G_2^1, G_1^2 * G_2^2}$$

is locally complete but may not be sound, since there is a case when G_1^1 and G_2^2 are true but G_2^1 and G_1^2 are false. We will propose a new split rule for disjunction. We do not transform the goal into a single subgoal set that keeps validity, but instead we will transform the goal into a set of subgoal sets such that at least one subgoal set keeps validity. We will say these rules are *selectively local complete*. For example, by our split rule, $F_1 * F_2 \vdash G_1^1 * G_2^1, G_1^2 * G_2^2$ will be transformed into four subgoal sets:

(A) $F_1 \vdash G_1^1$ and $F_1 \vdash G_1^2$ and $F_2 \vdash G_2^1, G_2^2$,

or (B) $F_1 \vdash G_1^2$ and $F_2 \vdash G_2^2$,

or (C) $F_1 \vdash G_1^1$ and $F_2 \vdash G_2^1$,

or (D) $F_1 \vdash G_1^1, G_1^2$ and $F_2 \vdash G_2^1$ and $F_2 \vdash G_2^2$.

Then we can show that at least one of these four cases is valid when $F_1 * F_2 \models G_1^1 * G_2^1, G_1^2 * G_2^2$.

(4) We explain an outline of our proof search algorithm.

Our proof search algorithm is based on the U-M-R-S strategy illustrated in the introduction. Given a goal entailment J, we start with a singleton subgoal set $\{J\}$. We repeatedly apply the following steps to a subgoal set to transform a goal entailment into subgoal entailments keeping validity. When the subgoal set becomes empty, we return Yes. If a new subgoal entailment is the same as that appeared already during computation, we finish this subgoal, since we can discharge this subgoal by the bud-companion relation in cyclic proofs. This algorithm is nondeterministic because of choice of the split rules at the step 6.

Step 1. Choose a subgoal entailment J from a subgoal set.

Step 2. Unfold atomic formulas of the root x in both antecedent and succedent of J, for some common root x in the antecedent and each disjunct of the succedent. If necessary, we use the factor rule to find a common root.

Step 3. Match the atomic formula $x \mapsto (\overrightarrow{t})$ of root x in both antecedent and succedent.

Step 4. Remove the same $x \mapsto (\overrightarrow{t})$ in both the antecedent and the succedent.

Step 5. If the antecedent has emp, do the following. If the succedent also has emp, then we finish this subgoal (this subgoal is valid) and remove it from the subgoal set. Otherwise we return No (this subgoal is invalid).

Step 6. Apply the split rule repeatedly until the spatial part of the antecedent in every entailment becomes atomic. According to our split rule, we will split it so that the allocated variables of the antecedent and each disjunct of the succedent coincide. If necessary, use the factor rule to divide an atomic formula so that the split rule is applicable.

This algorithm terminates since the set of entailments whose antecedent spatial part is atomic is finite up to variable renaming after some normalization. If the algorithm were not terminating, by executing the step 6 infinitely many time, we would have infinitely many such entailments, which leads to contradiction.

5 Logical System CSLID$^\omega$

This section defines our logical system CSLID$^\omega$ and shows its soundness.

5.1 Inference Rules

This subsection gives the set of inference rules.

We write $F[F']$ to explicitly display the subformula F' at a positive position in F. We say Φ is *equality-full* when Π contains $(\neq (\overrightarrow{y}, V \cup \overrightarrow{y} \cup \{\mathrm{nil}\}))$ where Φ is $\exists \overrightarrow{x} \exists \overrightarrow{y} \Uparrow (\Pi \wedge \Gamma)$ and $V = \mathrm{FV}(\Phi)$.

Standard or easy inference rules are given in Figure 1. The other inference rules are given in Figure 2. A set of rules is defined to be *selectively locally complete* if there is some locally complete rule in the set.

The rule (Factor) is sound, since $P(t)$ from $(Q(y) {-\!\!*}^{\mathrm{ind}} P(t)) * Q(y)$ is true by the definition of ${-\!\!*}^{\mathrm{ind}}$. For local completeness, the rule (Factor) lists up all possible cases for $Q(y)$ in the disjunction.

The rules ($\exists$ Amalg1, 2) amalgamate $\exists x$'s under some condition, which guarantees that existentials have the same values. The soundness and local completeness of these rules can be shown by using the definition of $t \downarrow$ and $t \Uparrow$.

The rule $(*)$ is a new split rule since it handles disjunction in the succedent. We explain the idea of our split rule. Consider $F_1 * F_2 \models G_1^1 * G_2^1, G_1^2 * G_2^2, G_1^3 * G_2^3, G_1^4 * G_2^4$ where $\mathrm{Alloc}(F_1) = \mathrm{Alloc}(G_1^i)$ and $\mathrm{Alloc}(F_2) = \mathrm{Alloc}(G_2^i)$ for $i =$

$$\frac{F \vdash \vec{G}}{F\theta \vdash \vec{G}\theta}\ (\text{Subst}) \qquad \frac{}{\mathrm{emp} \vdash \mathrm{emp}}\ (\text{emp}) \qquad \frac{}{F \vdash \vec{G}}\ (\text{Unsat})\ (F \text{ unsatisfiable})$$

$$\frac{F \vdash \vec{G}}{F' \wedge F \vdash \vec{G}}\ (\wedge L) \qquad \frac{F \vdash \vec{G}, \exists\vec{x}(G \wedge G')}{F \vdash \vec{G}, \exists\vec{x} G}\ (\wedge\text{Elim}) \qquad \frac{F \vdash \vec{G}, \exists\vec{x}(G \wedge \neg G)}{F \vdash \vec{G}}\ (\vee\text{Elim})$$

$$\frac{F \wedge F' \vdash \vec{G}, G}{F \wedge F' \vdash \vec{G}, G \wedge F'}\ (\wedge R) \qquad \frac{(x \uparrow \wedge F_1) * F_2 \vdash \vec{G}}{F_1 * F_2 \vdash \vec{G}}\ (\uparrow \text{Elim})\ (x \in \mathrm{Alloc}(F_2))$$

$$\frac{F \vdash \vec{G}}{F \vdash \vec{G}, G}\ (\vee R) \qquad \frac{F \vdash \vec{G}, G[w := t]}{F \vdash \vec{G}, \exists w G}\ (\exists R) \qquad \frac{F[x := t] \vdash \vec{G}[x := t]}{x = t \wedge F \vdash \vec{G}}\ (= L) \qquad \frac{F \vdash \vec{G}, G}{F \vdash \vec{G}, G \wedge t = t}\ (= R)$$

$$\frac{F * (F_1 \wedge x \downarrow) * F_2 \vdash \vec{G} \quad F * F_1 * (F_2 \wedge x \downarrow) \vdash \vec{G}}{F * ((F_1 * F_2) \wedge x \downarrow) \vdash \vec{G}}\ (\downarrow \text{Case } L) \qquad \frac{F \vdash \vec{G}, G}{F \vdash \vec{G}, \exists y \Uparrow (y \neq t \wedge G)}\ (\Uparrow R)\ (y \notin \mathrm{FV}(G))$$

$$\frac{F \vdash \vec{G}, \exists\vec{x}(G * (G_1 \wedge t \downarrow) * G_2), \exists\vec{x}(G * G_1 * (G_2 \wedge t \downarrow))}{F \vdash \vec{G}, \exists\vec{x}(G * ((G_1 * G_2) \wedge t \downarrow))}\ (\downarrow \text{Case } R) \qquad \frac{F * \mathcal{P}(t, \vec{t}) \vdash \vec{G}, G}{F * \mathcal{P}(t, \vec{t}) \vdash \vec{G}, G \wedge t \downarrow}\ (\downarrow R)$$

$$\frac{F \wedge F' \vdash \vec{G} \quad F \wedge \neg F' \vdash \vec{G}}{F \vdash \vec{G}}\ (\text{Case } L)$$

$$\frac{A(x, \vec{t}, \vec{z}) * F \vdash \vec{G} \quad (\text{for every definition clause } \exists\vec{z} A(x, \vec{t}, \vec{z}) \text{ of } P(x, \vec{t}))}{P(x, \vec{t}) * F \vdash \vec{G}}\ (\text{Pred } L)$$

$$\frac{F \vdash \vec{G}, \{G[\phi] \mid \phi \text{ is a definition clause of } P(x, \vec{t})\}}{F \vdash \vec{G}, G[P(x, \vec{t})]}\ (\text{Pred } R)\ (G[\ ]\ \forall\text{-free})$$

$$\frac{x \uparrow \wedge F \vdash \{G_i \mid i \in I\}}{F * x \mapsto (\vec{u}) \vdash \{G_i * x \mapsto (\vec{u}) \mid i \in I\}}\ (* \mapsto)$$

Fig. 1. Inference Rules 1

$1, 2, 3, 4$. Let I be $\{1, 2, 3, 4\}$. Then for any $I' \subseteq I$ we have $F_1 \models \{G_1^i \mid i \in I'\}$ or $F_2 \models \{G_2^i \mid i \in I - I'\}$. It is because $F_1 \not\models \{G_1^i \mid i \in I'\}$ and $F_2 \not\models \{G_2^i \mid i \in I - I'\}$ imply $F_1 * F_2 \not\models G_1^1 * G_2^1, G_1^2 * G_2^2, G_1^3 * G_2^3, G_1^4 * G_2^4$. For example, by taking I' to be $\{1, 2\}$, we have $F_1 \models G_1^1, G_1^2$ or $F_2 \models G_2^3, G_2^4$. The split rule is defined by picking up either of $F_1 \vdash \{G_1^i \mid i \in I'\}$ or $F_2 \vdash \{G_2^i \mid i \in I - I'\}$ for each $I' \subseteq I$ and taking them to be the assumptions. Then each of these rules becomes sound, by some property of propositional logic. Moreover at least one of these rule becomes locally complete, since either of $F_1 \models \{G_1^i \mid i \in I'\}$ or $F_2 \models \{G_2^i \mid i \in I - I'\}$ for all $I' \subseteq I$.

5.2 Proofs in CSLID$^\omega$

We define a proof in CSLID$^\omega$. It is the same as that in [9] except we use a slightly different form of global trace condition.

Definition 5.1 For CSLID$^\omega$, we define a *preproof* to be an ordinary proof figure by the inference rules with open assumptions. For a preproof, we consider a map (called a *bud-companion relation*) from the set of occurrences of open assumptions (called a *bud*) to the set of inner occurrences of sequents (called a *companion*). For CSLID$^\omega$, We define a *cyclic proof* to be a preproof with a

$$\dfrac{\begin{array}{l}F \vdash \overrightarrow{G}, \{G[\exists \overrightarrow{w}((\overrightarrow{\overline{Q_1(\overrightarrow{t}_1)}}, Q(y, \overrightarrow{w}) {-\!\!*}^{\mathrm{ind}} P(\overrightarrow{t})) * (\overrightarrow{\overline{Q_2(\overrightarrow{t}_2)}} {-\!\!*}^{\mathrm{ind}} Q(y, \overrightarrow{w})))] \mid \\ \overrightarrow{\overline{R(\overrightarrow{u})}} = (\overrightarrow{\overline{Q_1(\overrightarrow{t}_1)}} + \overrightarrow{\overline{Q_2(\overrightarrow{t}_2)}}),\ Q \in \mathrm{Dep}(P),\ \overrightarrow{Q}_2 \subseteq \mathrm{Dep}(Q),\ \overrightarrow{w} \text{ fresh}\}\end{array}}{F \vdash \overrightarrow{G}, G[\overrightarrow{\overline{R(\overrightarrow{u})}} {-\!\!*}^{\mathrm{ind}} P(\overrightarrow{t})]} \text{ (Factor) } (G[\ ]\ \forall\text{-free})$$

$$\dfrac{F \vdash \overrightarrow{G}, G[\exists x \Uparrow \Phi_1 * \exists x \Uparrow \Phi_2]}{F \vdash \overrightarrow{G}, G[\exists x \Uparrow (\Phi_1 * \Phi_2)]} \text{ (}\exists\text{ Amalg1) } (\exists x \Uparrow \Phi_1,\ \exists x \Uparrow \Phi_2 \text{ equality-full})$$

$$\dfrac{F \vdash \overrightarrow{G}, G[\exists x \Phi_1 * \exists x \Uparrow \Phi_2]}{F \vdash \overrightarrow{G}, G[\exists x(\Phi_1 * (x \Uparrow \wedge \Phi_2))]} \text{ (}\exists\text{ Amalg2) } \left(\begin{array}{l}x \in \mathrm{Cells}(\Phi_1),\ (x \neq \mathrm{FV}(\exists x \Uparrow \Phi_2)) \subseteq \Phi_1, \\ \exists x \Uparrow \Phi_2 \text{ equality-full}\end{array}\right)$$

$$\dfrac{F_1 \vdash \{G_1^i \mid i \in I'\} \text{ or } F_2 \vdash \{G_2^i \mid i \in I - I'\} \quad (\forall I' \subseteq I)}{F_1 * F_2 \vdash \{G_1^i * G_2^i \mid i \in I\}} \ (*)$$

Fig. 2. Inference Rules 2

bud-companion relation where each bud has a companion below it and there is some rule $(* \mapsto)$ between them.

Instead of the global trace condition in ordinary cyclic proof systems [9], CSLID$^\omega$ requires some $(* \mapsto)$ rule between a bud and its companion.

We can show the soundness theorem of CSLID$^\omega$ by using the fact that $|\mathrm{Dom}(h)|$ decreases upwardly by the rule $(* \mapsto)$.

Theorem 5.2 (Soundness) *If J is provable in* CSLID$^\omega$*, then J is valid.*

6 Proof Search Algorithm

This section gives the proof search algorithm to decide the provability of a given entailment. It will also be shown to decide the validity of a given entailment. First we define normal form, next define the algorithm, then we will show the partial correctness, and finally the termination of the algorithm by using normal form.

6.1 Normal Form

This section defines normal form.

In our proof search algorithm, a normal form appears as a bud in cyclic proofs. A normal form is obtained from an entailment such that the spatial part of its antecedent is a single cone, by transforming it into a simpler form keeping validity. The set of normal forms with d can be shown to be finite up to renaming. Since there is some d such that we can show that extended inductive predicates of depth $\leq d$ are sufficient for the algorithm, the termination of the algorithm will be proved by counting normal forms.

Definition 6.1 (Normal Form) For a given number d, an entailment J is called *normal* with d if J is of the form $Y \uparrow \wedge \Pi \wedge \Gamma \vdash \{\Phi_i \mid i \in I\}$ and Φ_i is of the form $\exists \overrightarrow{x}_i \exists \overrightarrow{y}_i \Uparrow (\Pi_i \wedge \Gamma_i)$ and by letting V be $\mathrm{FV}(J)$,

1. Γ is a single cone (single cone condition),
2. $Y + \mathrm{Alloc}(\Gamma) = V$ (variable condition),
3. $\mathrm{Roots}(\Phi_i)$ is defined (disjunct root condition),
4. $\mathrm{Alloc}(\Gamma) = \mathrm{Alloc}(\Phi_i)$ for every $i \in I$ (allocation condition),
5. $\overrightarrow{x}_i \subseteq \mathrm{Cells}(\Gamma_i)$ (disjunct existential condition),
6. Π is $(\neq V)$ (equality condition),
7. Π_i is $(\neq (\overrightarrow{x}_i \overrightarrow{y}_i, V + \{\overrightarrow{x}_i \overrightarrow{y}_i, \mathrm{nil}\})$ (disjunct equality condition),
8. if $i \neq j$, then $\Phi_i \not\equiv \Phi_j \theta$ for all variable renaming θ such that $\mathrm{Dom}(\theta) \cap \mathrm{FV}(\Gamma) = \emptyset$ (disjunct renaming condition),
9. $\mathrm{FV}(Y, \Pi) \subseteq \mathrm{FV}(\Gamma, (\Phi_i)_i)$ (antecedent variable condition),
10. $|\overrightarrow{Q}| \leq d$ for every predicate symbol $\overrightarrow{Q} \mathrel{-\!\!*}^{\mathrm{ind}} P$ in J (wand condition).

For example, the following is a normal form:

$$\{y, z\} \uparrow \wedge \neq (\{x, y, z\}) \wedge \mathrm{ls}(x, y) \vdash \exists w \Uparrow (w \neq \{x, y, z, \mathrm{nil}\} \wedge \mathrm{ls}(y, w) \mathrel{-\!\!*}^{\mathrm{ind}} \mathrm{ls}(x, z)).$$

6.2 Definition of Proof Search Algorithm

For a given entailment $A \vdash \overrightarrow{B}$ of quantifier-free symbolic heaps, our proof search algorithm returns Yes or No according to whether $A \vdash \overrightarrow{B}$ is valid or not.

The function MainLoop is called in the main function of the algorithm. Mainloop repeatedly executes the unfold-match-remove-split steps, to produce subgoals from a subgoal. When the same subgoal is generated as that generated already, by cyclic proof mechanism, this subgoal is discharged immediately. Because of a choice of the split rules, MainLoop is executed nondeterministically.

Let $k_{\max}$ be the maximum arity for predicate symbols in the original language.

We assume a satisfiability checking procedure for ψ by extending that for symbolic heaps given in [12]. This procedure is given in [32].

In our algorithm, we do the following trivial steps at several places and we omit their description for simplicity: (1) case analysis by $x = t \vee x \neq t$ for each variable x and term t, (2) case analysis by $x \downarrow \vee x \uparrow$, (3) transformation into the form of entailment $\psi \vdash \overrightarrow{\Phi}$, (4) removing unsatisfiable disjuncts, (5) removing a subgoal when its antecedent is unsatisfiable. For (1), the case analysis by $x = t \vee x \neq t$ in the antecedent means to transform $F \vdash \overrightarrow{G}$ into two subgoals $x = t, F \vdash \overrightarrow{G}$ and $x \neq t, F \vdash \overrightarrow{G}$. The case analysis by $x = t \vee x \neq t$ in the succedent means to transform $F \vdash (G_i)_i$ into a subgoal $F \vdash (G_i \wedge x = t, G_i \wedge x \neq t)_i$. For (2), the case analysis in the antecedent and the succedent is similarly defined to (1).

The proof search algorithm is defined in Algorithm 1.

Algorithm 1: Proof Search Algorithm

```
input  : quantifier-free symbolic heap $A \vdash \overrightarrow{B}$
output: Yes or No
Do case analysis of $=, \neq$ and $\downarrow, \uparrow$ on $A \vdash \overrightarrow{B}$ to obtain subgoal entailments $\overrightarrow{J}$.
for each $J$ in $\overrightarrow{J}$ do
    $\mathrm{d}_{\mathrm{wand}} := k_{\max} + |\mathrm{Alloc}(\text{antecedent of } J)|$.
    Call MainLoop$(J, \mathrm{d}_{\mathrm{wand}})$.
    if some nondeterministic computation of MainLoop returns Yes then
        continue
    else return No
    end if
end for
return Yes.
```

The input for the proof search algorithm is an entailment of quantifier-free symbolic heaps. First we do case analysis of $=, \neq$ and $\downarrow, \uparrow$ to produce subgoals $\overrightarrow{J}$. For example,

$$\mathrm{ls}(x,y) * \mathrm{ls}(y,z) \vdash \mathrm{ls}(x,z)$$

is transformed into subgoals, one subgoal of which is equivalent to

$$z \uparrow \wedge z = x \wedge z \neq y \wedge z \neq \mathrm{nil} \wedge \mathrm{ls}(x,y) * \mathrm{ls}(y,z) \vdash \mathrm{ls}(x,z) \wedge y \downarrow .$$

MainLoop is nondeterministically executed. When some nondeterministic computation returns Yes, J is valid and we go to the next subgoal. If every nondeterministic computation returns No, J is invalid and we return No. When all the subgoals are solved, we return Yes.

The function MainLoop is defined in Algorithm 2.

The input J of MainLoop is a goal entailment J and the input d is the depth for the factor procedure. First we set S to be $\{(J, \emptyset)\}$. S is a set of pairs of a subgoal and a history (a set of entailments that appear already). In the while loop, we take a subgoal J and a history H from the set S. If J appears already, we can discharge J by cyclic proof mechanism, and go to the next subgoal. We add J to the history H. If J does not have common roots on both sides, then we call the factor procedure with depth d. The factor procedure applies the factor rule with the condition $|\overrightarrow{Q_1(\overrightarrow{t}_1), Q(y, \overrightarrow{w})}| \leq d$ and $|\overrightarrow{Q_2(\overrightarrow{t}_2)}| \leq d$. Then we do the unfold-match-remove steps, namely, we unfold predicates with the common root x in the antecedent and each disjunction of the succedent, and we match $x \mapsto (\overrightarrow{t})$ on both sides of the antecedent and the succedent, and we remove the same $x \mapsto (\overrightarrow{t})$ from the antecedent and the succedent. Then we check termination condition as follows when the spatial part of the antecedent is emp: if the spatial part of some disjunct in the succedent is emp, we discharge this subgoal (J is valid), and otherwise we return No (J is invalid). Then we apply the split procedure. In the split procedure, first we apply the factor rule to divide an atomic formula if necessary, next we apply ($\exists$Amalg1) and ($\exists$Amalg2) for dividing existential scopes if necessary, and then we apply the split rule under

Algorithm 2: Function MainLoop

```
input  : goal entailment $J$, maximum inductive wand depth $d$
output: Yes or No
$S := \{J\}$. $H := \emptyset$.
while $S \neq \emptyset$ do
    Choose $J \in S$.
    $S := S - \{J\}$.
    if there are some $J' \in H$ and $\theta$ such that $J'\theta \equiv J$ then continue
    $H := H + \{J\}$
    if $J$ does not have common roots then
        apply the factor procedure with depth $d$ to $J$.
    end if
    Do unfold-match-remove steps to $J$.
    if antecedent of $J$ has emp then
        if succedent of $J$ has emp then continue
        else return No
    end if
    Apply the split procedure repeatedly to $J$ to obtain a set $G$ of subgoal sets
  with a single cone.
    Nondeterministically choose a subgoal set $R \in G$.
    for each $J$ in $R$ do
        Normalize $J$.
        $S := S \cup \{J\}$.
    end for
end while
return Yes
```

the condition $\mathrm{Alloc}(F_j) = \mathrm{Alloc}(G^i_j)$ for $j = 1, 2$ and $i \in I$. Since the split rules at this step are selectively locally complete, we try all the split rules and we produce a set G of subgoal sets instead of a single subgoal set. We repeat until subgoals become those with a single cone. At least one subgoal set in G keeps validity. Hence we nondeterministically continue computation for each subgoal set R in G. Then we transform each subgoal in R into normal form and put it into the subgoal set S.

We can show the partial correctness of the algorithm with Yes, by checking each step consists of application of inference rules. We will discuss the case with No in the completeness proof later. Note that we can transform a cyclic proof produced by the algorithm such that the companion may not be below some bud into a cyclic proof such that the companion is below any bud, by expanding each bud by the companion some times and finding a repetition on each path.

Lemma 6.2 (Partial Correctness) *If the algorithm returns Yes, then the input entailment is provable.*

6.3 Termination

This subsection shows the termination of the algorithm.

Since a normal form during the loop has the maximum depth of inductive wands, the number of normal forms up to variable renaming is proved to be finite.

Lemma 6.3 *The set of normal forms with d up to variable renaming is finite.*

We can show the termination by using the finiteness.

Lemma 6.4 (Termination) *(1) Every nondeterministic computation of MainLoop terminates.*

(2) The proof search procedure terminates.

The proof of the previous lemma evaluates the length of the history H used in the algorithm. By using it, we can show time complexity of the algorithm.

Proposition 6.5 *The time complexity of the proof search algorithm is nondeterministic double exponential time.*

7 Completeness of CSLID$^\omega$

This section shows the completeness of CSLID$^\omega$ by using the algorithm.

By using the properties of each step in the algorithm, we can show that for a valid input, some nondeterministic computation does not return No.

Lemma 7.1 *(1) Each step in the proof search algorithm except the application of the split rule and the factor rule transforms a valid entailment into valid entailments.*

(2) In the proof search algorithm there is some nondeterministic computation in which every application of the split rule and the factor rule is locally complete.

(3) If a valid entailment is given to MainLoop, some nondeterministic computation does not return No.

Finally we can prove the completeness of CSLID$^\omega$.

Theorem 7.2 (Completeness) *(1) The system* CSLID$^\omega$ *is complete. Namely, if a given entailment J is valid, then it is provable in* CSLID$^\omega$*.*

(2) The proof search algorithm decides the validity of a given entailment. Namely, For a given input J, the proof search algorithm returns Yes when the input is valid, and it returns No when the input is invalid.

Proof. (1) Assume J is valid in order to show J is provable in CSLID$^\omega$. When we input J to the algorithm, by Lemma 7.1 (3), in each case of calling MainLoop, some nondeterministic computation does not return No. By Lemma 6.4 (1), the nondeterministic computation returns Yes. Hence the algorithm returns Yes. By Lemma 6.2, J is provable.

(2) Assume J is valid, in order to show the algorithm with input J returns Yes. In the same way as (1), the algorithm is shown to return Yes.

Assume J is invalid, in order to show the algorithm with input J returns No. By Lemma 6.4 (2), the algorithm terminates. Assume that it returns Yes, in order to show contradiction. By Lemma 6.2, J is provable. By Theorem 5.2, J is valid, which leads to contradiction. □

8 Implementation and Experiments

This section explains our entailment checker Cycomp, which is an implementation of our proof search algorithm. Cycomp is implemented in OCaml with about 7600 lines of codes (including the internally-called satisfiability checker and some optimization). The core part is an implementation of the pseudocode given in a detailed version of this paper [32]. The test problems for evaluating Cycomp and the definitions of inductive predicates used in the problems are presented in Table 1 and Table 2, respectively.

The table also compares Cycomp with the other two state-of-the-art entailment solvers for separation logic with general inductively defined predicates: Songbird and Cyclist. Songbird searches structural induction proofs synthesizing lemmas which would be induction hypotheses. We used the latest version of Songbird (called SLS [28]), which can solve most of the valid entailments from SL-COMP [27]. Cyclist [33] is based on a proof search procedure with the Unfold-Match strategy on cyclic proofs. Our test was done with the option of Cyclist that enables lemma synthesis. OutScope in the Songbird column about the problem 4 means that the format of the problem is out of the syntactic restriction of Songbird, namely the problem contains multiple conclusions. Times with (UN) and (IC) in that column mean that Songbird answered "Unknown" and an incorrect answer with that time, respectively.

The test was done on a laptop PC with a 1.60GHz Intel(R) Core(TM) i5-8250U CPU, 8GB memory, and Linux Mint 19. The inputs were executed with 600 seconds timeout setting.

In general, for valid problems, the performance of Cycomp strongly depends on the numbers of inductive predicates and variables that appear in an input entailment. These numbers cause increasing of the number of succedents after applying the factor rule, then the number of the case analysis for ($*$)-rule drastically increases, since it requires 2^n cases for an subgoal entailment with n-succedents. Our implementation contains some simple optimization processes to reduce this increase as much as possible. With this optimization, Cycomp quickly shows problems with small numbers, such as the problems 1, 2, and 3. The problem 11 is obtained from the problem 12 by substituting nil for the variables p and n. Cycomp can show the problem 11 faster than the problem 12, since the number of variables are decreased by the substitution. (Interestingly, Cyclist and Songbird have the opposite results.) However, for more complicated valid problems such as the problems from 21 to 25, Cycomp causes time out. In order to obtain a more efficient procedure, it would be important to introduce suitable heuristics to handle numbers of succedents. Supporting the lemma synthesis mechanism would be a possible direction.

For invalid problems, Cycomp explores all branches including back-tracking of the ($*$)-split rule and finally answers "Invalid" when all the branches are finished with failure. Although this mechanism may potentially take time depending on problems, Cycomp can finish quickly if it finds a contradiction of each branch at an earlier stage, as our experimental results of the problems 5, 7, and 26–30 show. For these invalid problems, Cyclist timed out and Songbird

No.	Problem	Status	**Cycomp**	Cyclist	Songbird
1	$\texttt{ls}(x,y) * \texttt{list}(y) \vdash \texttt{list}(x)$	Valid	0.020	0.072	0.098
2	$\texttt{ls}(x1,x2) * \texttt{ls}(x2,x3) \vdash \texttt{ls}(x1,x3)$	Valid	0.121	0.075	0.071
3	$\texttt{ls}(x1,x2) * \texttt{ls}(x2,x3) * \texttt{ls}(x3,x4) \vdash \texttt{ls}(x1,x4)$	Valid	0.446	0.546	0.102
4	$\texttt{ls}(x,y) \vdash \texttt{lsO}(x,y), \texttt{lsE}(x,y)$	Valid	0.258	Timeout	OutScope
5	$\texttt{ls}(x,y) \vdash \texttt{lsO}(x,y)$	Invalid	0.122	Timeout	(UN)6.345
6	$\texttt{lsO}(x,y) * \texttt{lsE}(y,z) \vdash \texttt{ls}(x,z)$	Valid	1.953	1.534	0.056
7	$\texttt{ls}(x,y) \vdash \texttt{lsa}(x,y,y)$	Invalid	0.239	Timeout	(IC) 6.251
8	$\texttt{ls}(x,z) * \texttt{ls}(z,x) \vdash \texttt{lsa}(x,x,z)$	Valid	0.187	14.241	0.158
9	$\texttt{lsa}(x,x,y) \vdash \texttt{lsa}(y,y,x)$	Valid	0.819	Timeout	(IC) 0.040
10	$h \mapsto (p,z) * \texttt{dll}(z,h,n,t) \vdash \texttt{dll}(h,p,n,t)$	Valid	15.157	0.019	0.031
11	$\texttt{dll}(h,\text{nil},\text{nil},t) \vdash \texttt{dllr}(t,\text{nil},\text{nil},h)$	Valid	0.342	Timeout	13.320
12	$\texttt{dll}(h,p,n,t) \vdash \texttt{dllr}(t,n,p,h)$	Valid	225.930	0.257	0.099
13	$x \mapsto (y,a) * \texttt{slk1}(a,y) * \texttt{slk2}(y,z) \vdash \texttt{slk2}(x,z)$	Valid	1.167	0.016	0.212
14	$x \mapsto (y,a) * \texttt{slk1}(a,b) * \texttt{slk1}(b,y) * \texttt{slk2}(y,\text{nil}) \vdash \texttt{slk2}(x,\text{nil})$	Valid	2.154	4.421	0.136
15	$\texttt{bpath}(x,y) * \texttt{bpath}(y,z) \vdash \texttt{bpath}(x,z)$	Valid	0.422	0.169	0.057
16	$\texttt{bpath}(x,y) \vdash \texttt{bts}(x,y)$	Valid	1.309	0.124	0.081
17	$\texttt{bts}(x,\text{nil}) \vdash \texttt{bt}(x)$	Valid	0.117	0.280	0.591
18	$\texttt{bt}(x) \vdash \texttt{bts}(x,\text{nil})$	Valid	0.086	0.256	0.630
19	$\texttt{bts}(x,y) * \texttt{bt}(y) \vdash \texttt{bt}(x)$	Valid	2.457	0.398	1.060
20	$\texttt{bpath}(x,y) * \texttt{bts}(y,\text{nil}) \vdash \texttt{bt}(x)$	Valid	1.016	20.046	0.959
21	$\texttt{dll}(h,p,z,w) * \texttt{dlist}(z,w,t) \vdash \texttt{dlist}(h,p,t)$	Valid	Timeout	0.066	0.051
22	$\texttt{dll}(h,\text{nil},z,u) * \texttt{dll}(z,u,\text{nil},t) \vdash \texttt{dll}(h,\text{nil},\text{nil},t)$	Valid	Timeout	2.457	0.071
23	$\texttt{slk2}(x,y) * \texttt{slk2}(y,\text{nil}) \vdash \texttt{slk2}(x,\text{nil})$	Valid	Timeout	0.182	0.117
24	$\texttt{bts}(x,y) * \texttt{bts}(y,\text{nil}) \vdash \texttt{bts}(x,\text{nil})$	Valid	Timeout	0.749	0.299
25	$\texttt{bpath}(x,y) * y \mapsto (l,r) * \texttt{bt}(l) * \texttt{bts}(r,\text{nil}) \vdash \texttt{bts}(x,\text{nil})$	Valid	Timeout	1.394	0.585
26	$\texttt{ls}(x,x) * \texttt{list}(y) \vdash \texttt{list}(x)$	Invalid	0.018	Timeout	(UN)0.342
27	$\texttt{ls}(x1,x2) * \texttt{ls}(x2,x3) \vdash \texttt{ls}(x1,x1)$	Invalid	0.038	Timeout	(UN)0.041
28	$\texttt{lsO}(x,y) * \texttt{lsE}(y,z) \vdash \texttt{ls}(x,x)$	Invalid	0.333	Timeout	(UN)7.461
29	$\texttt{dll}(h,\text{nil},\text{nil},t) \vdash h \mapsto (\text{nil},\text{nil})$	Invalid	0.029	Timeout	(UN)0.039
30	$\texttt{dll}(h,\text{nil},\text{nil},t) \vdash \texttt{dllr}(h,\text{nil},\text{nil},t)$	Invalid	0.055	Timeout	(UN)8.863

Table 1. Experimental results

almost answered "Unknown", since they are not decision procedures. It would be an advantage of Cycomp against these existing solvers.

9 Conclusion

We have proposed the cyclic proof system CSLID$^\omega$ for symbolic heaps with cone inductive definitions, and have proved its soundness theorem and its completeness theorem, and have given the proof search algorithm that decides the validity of a given entailment. Furthermore we have implemented a prototype system for the algorithm and have presented experimental results.

Future work would be to extend ideas in this paper to other systems, in particular, a system with arrays.

Acknowledgments

We would like to thank Prof. Kazushige Terui for valuable discussions. This is partially supported by Core-to-Core Program (A. Advanced Research Networks) of the Japan Society for the Promotion of Science.

Singly-linked list	$\texttt{list}(x) := x \mapsto (\mathrm{nil}) \vee \exists z(x \mapsto (z) * \texttt{list}(z))$
Lseg with odd length	$\texttt{lsO}(x,y) := x \mapsto (y) \vee \exists z(x \mapsto (z) * \texttt{lsE}(z,y))$
Lseg with even length	$\texttt{lsE}(x,y) := \exists z(x \mapsto (z) * \texttt{lsO}(z,y))$
Lseg with allocated cell	$\texttt{lsa}(x,y,z) := x = z \wedge x \mapsto (y) \vee \exists w(x = z \wedge x \mapsto (w) * \texttt{lsa}(w,y,w))$
	$\vee \exists w(x \mapsto (w) * \texttt{lsa}(w,y,z))$
Doubly-linked list	$\texttt{dlist}(h,p,t) := h = t \wedge h \mapsto (p,\mathrm{nil}) \vee \exists z(h \mapsto (p,z) * \texttt{dlist}(z,h,t))$
Reversed dll	$\texttt{dllr}(h,p,n,t) := h = t \wedge h \mapsto (n,p) \vee \exists z(h \mapsto (z,p) * \texttt{dllr}(z,h,n,t))$
Skip list (1st level)	$\texttt{slk1}(a,b) := a \mapsto (\mathrm{nil},b) \vee \exists c(a \mapsto (\mathrm{nil},c) * \texttt{slk1}(c,b))$
Skip list (2nd level)	$\texttt{slk2}(x,y) := x \mapsto (y,y) \vee \exists z,a(x \mapsto (z,z) * \texttt{slk2}(z,y))$
	$\vee \exists z,a(x \mapsto (z,a) * \texttt{slk1}(a,z) * \texttt{slk2}(z,y))$
Binary tree	$\texttt{bt}(x) := x \mapsto (\mathrm{nil},\mathrm{nil}) \vee \exists l(x \mapsto (l,\mathrm{nil}) * \texttt{bt}(l)) \vee \exists r(x \mapsto (\mathrm{nil},r) * \texttt{bt}(r))$
	$\vee \exists l,r(x \mapsto (l,r) * \texttt{bt}(l) * \texttt{bt}(r))$
Binary tree segment	$\texttt{bts}(x,y) := x \mapsto (y,\mathrm{nil}) \vee x \mapsto (\mathrm{nil},y) \vee \exists l(x \mapsto (l,\mathrm{nil}) * \texttt{bts}(l,y))$
	$\vee \exists r(x \mapsto (\mathrm{nil},r) * \texttt{bts}(r,y)) \vee \exists l,r(x \mapsto (l,r) * \texttt{bt}(l) * \texttt{bts}(r,y))$
	$\vee \exists l,r(x \mapsto (l,r) * \texttt{bts}(l,y) * \texttt{bt}(r))$
Path in binary-tree	$\texttt{bpath}(x,y) := x \mapsto (\mathrm{nil},y) \vee x \mapsto (y,\mathrm{nil}) \vee \exists z(x \mapsto (z,\mathrm{nil}) * \texttt{bpath}(z,y))$
	$\vee \exists z(x \mapsto (\mathrm{nil},z) * \texttt{bpath}(z,y))$

Table 2. Definitions of inductive predicates

References

1. T. Antonopoulos, N. Gorogiannis, C. Haase, M. Kanovich, and J. Ouaknine, Foundations for Decision Problems in Separation Logic with General Inductive Predicates, In: Proceedings of FoSSaCS 2014, *LNCS* 8412 (2014) 411–425.
2. S. Berardi and M. Tatsuta, Classical System of Martin-Lof's Inductive Definitions is not Equivalent to Cyclic Proof System, In: Proceeding of FoSSaCS 2017, *LNCS* 10203 (2017) 301–317.
3. S. Berardi and M. Tatsuta, Equivalence of Inductive Definitions and Cyclic Proofs under Arithmetic, In: *Proceedings of LICS2017* (2017) 1–12.
4. R. Brochenin, S. Demri, and E. Lozes, On the Almighty Wand, In: *Proceedings of CSL 2008* (2008) 323–338.
5. B.-Y. .E. Chang and X. Rival, Relational Inductive Shape Analysis, In: *Proceedings of POPL 2008* (2008) 247–260.
6. J. Berdine, C. Calcagno, P. W. O'Hearn, A Decidable Fragment of Separation Logic, In: Proceedings of FSTTCS 2004, *LNCS* 3328 (2004) 97–109.
7. J. Berdine, C. Calcagno, and P. W. O'Hearn, Symbolic Execution with Separation Logic, In: Proceedings of APLAS 2005, *LNCS* 3780 (2005) 52–68.
8. J. Brotherston, Formalised Inductive Reasoning in the Logic of Bunched Implications, In: Proceedings of SAS 2007, *LNCS* 4634 (2007), 87–103.
9. J. Brotherston, A Simpson, Sequent calculi for induction and infinite descent, *Journal of Logic and Computation 21* (6) (2011) 1177–1216.
10. J. Brotherston, D. Distefano, and R. L. Petersen, Automated cyclic entailment proofs in separation logic, In: *Proceedings of CADE-23* (2011) 131–146.
11. J. Brotherston, N. Gorogiannis, and R. L. Petersen, A Generic Cyclic Theorem Prover, In: Proceedings of APLAS 2012, *LNCS* 7705 (2012) 350–367.
12. J. Brotherston, C. Fuhs, N. Gorogiannis, and J. Navarro Pérez, A Decision Procedure for Satisfiability in Separation Logic with Inductive Predicates, *In: Proceedings of CSL-LICS'14* (2014) Article 25.

13. J. Brotherston, N. Gorogiannis, M. Kanovich, and R. Rowe, Model checking for symbolic-heap separation logic with inductive predicates, In: *Proceedings of POPL 2016* (2016) 84-96.
14. W. Chin, C. David, H. Nguyen, and S. Qin, Automated Verification of Shape, Size and Bag Properties via User-Defined Predicates in Separation Logic, In *Science of Computer Programming* 77 (9) (2012) 1006–1036.
15. D. Chu, J. Jaffar, and M. Trinh, Automatic Induction Proofs of Data-Structures in Imperative Programs, In: *Proceedings of PLDI 2015* (2015) 457–466.
16. B. Cook, C. Haase, J. Ouaknine, M. J. Parkinson, J. Worrell, Tractable reasoning in a fragment of Separation Logic, In: Proceedings of CONCUR'11, *LNCS* 6901 (2011) 235–249.
17. C. Enea, V. Saveluc, M. Sighireanu, Compositional invariant checking for overlaid and nested linked lists, In: Proceeding of ESOP 2013, *LNCS* 7792 (2013) 129–148.
18. C. Enea, O. Lengál, M. Sighireanu, T. Vojnar, Compositional Entailment Checking for a Fragment of Separation Logic, In: Proceedings of APLAS 2014, *LNCS* 8858 (2014) 314–333.
19. R. Iosif, A. Rogalewicz, and J. Simacek, The Tree Width of Separation Logic with Recursive Definitions, In: Proceedings of CADE-24, *LNCS* 7898 (2013) 21–38.
20. R. Iosif, A. Rogalewicz, and T. Vojnar, Deciding Entailments in Inductive Separation Logic with Tree Automata, In: Proceedings of ATVA 2014, *LNCS* 8837 (2014) 201–218.
21. J. Katelaan, C. Matheja, and F. Zuleger, Effective Entailment Checking for Separation Logic with Inductive Definitions, In: Proceedings of TACAS 2019, *LNCS* 11428 (2019) 319–336.
22. J. Navarro Pérez and A. Rybalchenko, Separation logic modulo theories, In: Proceedings of APLAS 2013, *LNCS 8301* (2013) 90-106.
23. R. Piskac, T. Wies, and D. Zufferey, Automating Separation Logic Using SMT, In: Proceedings of CAV 2013, *LNCS 8044* (2013) 773-789.
24. R. Piskac, T. Wies, and D. Zufferey, Automating separation logic with trees and data, In: Proceedings of CAV 2014, *LNCS 8559* (2014) 711–728.
25. J.C. Reynolds, Separation Logic: A Logic for Shared Mutable Data Structures, In: *Proceedings of Seventeenth Annual IEEE Symposium on Logic in Computer Science (LICS2002)* (2002) 55–74.
26. A. Simpson, Cyclic Arithmetic is Equivalent to Peano Arithmetic, In: *Proceedings of FoSSaCS 2017* (2017) 283–300.
27. SL-COMP2014, `https://www.irif.fr/~sighirea/slcomp14/`.
28. SLS: Songbird+Lemma Synthesis, `https://songbird-prover.github.io/lemma-synthesis/`.
29. Q. Ta, T. Le, S. Khoo, and W. Chin, Automated Mutual Induction in Separation Logic, In: Proceedings of FM 2016, *LNCS* 9995 (2016) 659–676.
30. Q. Ta, T. Le, S.. Khoo, and W. Chin, Automated Lemma Synthesis in Symbolic-Heap Separation Logic, In: Proceedings of POPL 2018, (2018)
31. M. Tatsuta and D. Kimura, Separation Logic with Monadic Inductive Definitions and Implicit Existentials, In: Proceedings of APLAS 2015, *LNCS* 9458 (2015) 69–89.
32. M. Tatsuta, K. Nakazawa, and D. Kimura, Completeness of Cyclic Proofs for Symbolic Heaps, `https://arxiv.org/abs/1804.03938`, (2018).
33. The Cyclist Framework and Provers, `http://www.cyclist-prover.org/`.