Decidable entailment checking for concurrent separation logic with fractional permissions

Yeonseok Lee, Koji Nakazawa

We propose a subsystem of concurrent separation logic with fractional permissions introduced by Brotherston et al. Separation logic is an extension of Hoare logic that reasons about programs using shared mutable data. This logic has separating conjunction asserting that its subformulas hold for separated (disjoint) parts in the heap. Fractional permissions manage access permission of shared resources between concurrent threads. Brotherston et al. introduced an extension of concurrent separation logic with fractional permissions, but they did not discuss the decidability of logic. The heart of this paper is restricting formulas of the system to symbolic heaps. We argue that our system has sufficient expressiveness by showing some examples. We prove the decidability of the entailment checking for our system by normalization of formulas. We eliminate permissions by normalization, and therefore we can reduce the entailment checking problem to the existing decidable one.

1 Introduction

1.1 Background

Hoare triples are of the form $\{P\}\ C\ \{Q\}$, where P is the precondition and Q is the postcondition for the program C, respectively. Separation Logic [20] based on Hoare logic enables modular proofs of programs which operate pointers by reasoning separated (disjoint) portion of heap via *separating conjunction* ($*$). For example, $P * R$ means that P and R hold in two disjoint parts, respectively. A key point of this logic is the Frame rule

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}},$$

making local reasoning possible. We can check that the condition R can coexist if the program C does not affect R. This characteristic can increase scalability for more complex problems. There are many verification tools whose logical base is Separation Logic, such as Infer by *Facebook* [13], Smallfoot [3], VeriFast [17] and Space Invader [15].

Concurrent separation logic is an extension of separation logic designed to prove concurrent programs [8]. The Parallel rule

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}$$

deals multiple threads sharing the same memory. This rule says that two threads C_1 and C_2 run concurrently on *disjoint* memory resources $P_1 * P_2$, then after two threads terminate safely, the result becomes $Q_1 * Q_2$.

Mostly, threads in concurrent programs share some resources, and we have to manage permission of the resources to be *writable* or *read-only* on them. If two or more threads want to update some shared variables and write over them disorderly, then a disastrous situation of *data-race* can happen. One of the solutions is using *fractional permissions* [6]. If a permission value is 1, then the thread can write. If a permission value is less than 1 and greater than 0, then the thread can only read. For the permis-

---

有理数値で表現した権限を持つ並行分離論理の決定可能なエンテイルメントチェック.

Yeonseok Lee, Koji Nakazawa, 名古屋大学情報学研究科, Graduate School of Informatics, Nagoya University.

sion values 0.999 and 0.001, things that the threads can do are the same; *read-only*.

Permission is managed by splitting or merging permission values. To merge permission values, we use *weak separating conjunction* ($\circledast$) which also holds in overlapped formulas [9]. This means that heaps are overlapped and they must agree on the data value and permissions at the overlapping locations are merged [6][9].

Brotherston et al. [9] say that splitting a permission value is easy, but joining one is not. The following is an example in [9]. Let A be the formula $x \mapsto a \vee y \mapsto b$. $A^{1.0} \models A^{0.5} \circledast A^{0.5}$ holds. Intuitively, this formula means that a writable heap can be split into two read-only heaps. However, $A^{0.5} \circledast A^{0.5} \not\models A^{1.0}$. The LHS can be satisfied when $(x \mapsto a)^{0.5}$ and $(y \mapsto b)^{0.5}$. However, RHS requires $(x \mapsto a)^{1.0}$ or $(y \mapsto b)^{1.0}$.

To solve this problem, we should denote a heap saving information that two copied heaps come from the same heap. Brotherston et al. introduced *nominal label* of Hybrid Logic [5] to solve the problem. We can denote a specific heap by labeling the formula A with the label α and add permissions when we prove concurrent programs [9]. Therefore, $A^{0.5} \circledast A^{0.5} \not\models A^{1.0}$, but $(A \wedge \alpha)^{0.5} \circledast (A \wedge \alpha)^{0.5} \models (A \wedge \alpha)^{1.0}$.

1.2 Motivation

Entailment checking problem is important for every Hoare-style verification procedure. Given two formulas A and B, this procedure checks $A \models B$, i.e., whether every model of A is also a model of B or not. In general the entailment checking problem is undecidable [1], and there are papers showing *decidable* entailment checking with some restrictions of predicates [16][4][18]. Brotherston et al. did not discuss decidability of entailment checking problem of their system in [9].

Formulas of [16][4][18] are restricted to the *symbolic heaps*. The symbolic heaps are formulas of the form $\Pi \wedge \Sigma$ where Π is a pure (boolean) formula and Σ is a $*$-combination of heap predicates (spatial formulas). For convenience, we write $\Pi \,¦\, \Sigma$ for $\Pi \wedge \Sigma$. These formulas make the analogy with the in-place aspect of concrete heaps apparent [4] and serve as the basis of various automated verification tools such as Infer [13] and Verifast [17], etc.

1.3 Contributions

Our research proposes a subsystem SL_{LP}^{SH} of the system in [9] by restricting formulas to *symbolic heaps*, and proves *decidablity* of the entailment checking in the system. We restrict formulas in [9] to symbolic heaps by the equivalence $\Sigma \wedge \alpha \equiv @_\alpha \Sigma \,¦\, \alpha$. Here, $@_\alpha \Sigma$ means Σ is true in the heap denoted by the label α [9]. That is, we rewrite labeled spatial formula $(\Sigma \wedge \alpha)$ as a corresponding symbolic heap $(@_\alpha \Sigma \,¦\, \alpha)$.

We prove two concurrent programs to show expressiveness of our subsystem. The first program has two threads that read list segment $\mathtt{ls}(x, y)$ in parallel. The second one reads two different list segments $\mathtt{ls}(x, y)$ and $\mathtt{ls}(a, b)$ and there are nested concurrent threads.

We do entailment checking after the normalization of symbolic heaps. Our normalization procedure executes calculating permission values of each label. A normal form is $\circledast$-free and each label appears at most once. For instance, let us check an entailment

$$\Pi \,¦\, (\alpha^{0.2} \circledast \alpha^{0.3}) * \Sigma \stackrel{?}{\vdash} \Pi' \,¦\, \alpha^{0.1} \circledast (\alpha^{0.4} * \Sigma'),$$

where Σ and Σ' are label-free. After normalizing the entailment, we get the normal form

$$\Pi \,¦\, \alpha^{0.5} * \Sigma \vdash \Pi' \,¦\, \alpha^{0.5} * \Sigma'.$$

Then, we check the entailment relations between the normal forms. We show that our entailment problem is *decidable* by reducing our one to existing decidable one [16][21].

2 CSL with labels and permissions

We recall the system SL_{LP} in [9]. Here LP stands for Label and Permissions.

Permission values. Permission values are rational numbers π such that $0 < \pi \leqslant 1$ holds [7]. $\mathtt{Perm}$ is the set of permission values. Intuitively, $\pi = 1$ means *writable* permission, that is, mutation and deallocation of the value are allowed. $0 < \pi < 1$ means *read-only* permission, that is, only referring to the value is allowed and mutation or deallocation of the value is not allowed.

2.1 Syntax of SL_{LP}

Definition 2.1 (Formulas of SL_{LP})**.**

$\mathtt{Var} = \{x, y, ...\}$ is the set of variables. $\mathtt{Label} = \{\alpha, \beta, \gamma, ...\}$ is the set of labels. $FV(B)$ is a set of free variables in B.

The terms and formulas of SL_{LP} are defined as follows.

$t ::= \mathtt{nil} \mid x$	terms
$B ::= t = s$	pure atoms
$\mid \neg B \mid B \wedge B$	Boolean operations
$\mid \mathtt{emp} \mid x \mapsto y \mid \mathtt{ls}(x, y)$	spatial atoms
$\mid B^{\pi} \mid \alpha \mid @_{\alpha} B$	perms and labels
$\mid B * B \mid B \circledast B$	separating conjunction
$\mid B \mathrel{-\!\!*} B \mid B \mathrel{-\!\!\circledast} B$	separating implication

Pure atoms. *Pure atoms* deal with equalities and disequalities between variables.

Spatial atoms. $\mathtt{emp}$ means empty heap. $x \mapsto y$ means points-to fact, that is, x points to a cell containing y. $\mathtt{ls}(x, y)$ is linked list segment from x to y, which is maybe *cyclic*.

Perms and Labels. B^{π} means "π share" of the formula B. $@_{\alpha} B$ means B is true in the heap denoted by the label α.

Separating conjunction. $*$ is the separating conjunction, which means that heaps are *disjoint*. $\circledast$ is *weak* separating conjunction different from $*$ in that this can also describe *overlapped* heaps. $\circledast$ also adds permissions at overlapping heaps, e.g. $\alpha^{0.5} \circledast \alpha^{0.5} \models \alpha^{1.0}$.

Separating implication. Separating implication (magic wand) asserts that to extend the heap with a disjoint part satisfying its first argument results in a heap satisfying its second argument [20]. $\mathrel{-\!\!*}$ and $\mathrel{-\!\!\circledast}$ are the implications adjoint to $*$ and $\circledast$.

2.2 Semantics of SL_{LP}

Permission heap model (s, h, ρ). A stack is a function $s : \mathtt{Var} \to \mathtt{Val}$. $\mathtt{Val} = \mathbb{N}$ is the set of values. A stack s is extended to terms by $s(\mathtt{nil}) = 0$.

A permission heap (p-heap) is a finite partial function $h : \mathtt{Loc} \to_{fin} \mathtt{Val} \times \mathtt{Perm}$ from locations to permission-value pairs. $\mathtt{Loc}$ is the set of addressable locations. We suppose that $\mathtt{Loc} \subset \mathtt{Val}$ and $0 \notin \mathtt{Loc}$. $\mathtt{PHeap}$ is the set of all p-heaps.

We write $dom(h)$ for the domain of h, that is, the set of addressable locations where h is defined. We define the multiplication $\pi \bullet h$ of a permission heap h by a permission value π by

$$(\pi \bullet h)(l) = (v, \pi \times \pi') \quad \text{if} \quad h(l) = (v, \pi').$$

A valuation is a function $\rho : \mathtt{Label} \to \mathtt{PHeap}$. Here, we assume that different labels denote disjoint heaps. That is, if $\alpha \neq \beta$ then $dom(\rho(\alpha)) \cap dom(\rho(\beta)) = \varnothing$.

Strong heap composition. Two permission heaps h_1 and h_2 are *disjoint* when $dom(h_1) \cap dom(h_2) = \varnothing$. The strong composition $h_1 \circ h_2$ of two disjoint permission heaps h_1 and h_2 is defined as

$$(h_1 \circ h_2)(l) = \begin{cases} h_1(l) & \text{if } l \notin dom(h_2) \\ h_2(l) & \text{if } l \notin dom(h_1). \end{cases}$$

If h_1 and h_2 are not *disjoint* then $h_1 \circ h_2$ is undefined.

Weak heap composition. Two permission heaps h_1 and h_2 are *compatible* when for all $l \in dom(h_1) \cap dom(h_2)$ such that $h_1(l) = (v_1, \pi_1)$ and $h_2(l) = (v_2, \pi_2)$, $v_1 = v_2$ and $\pi_1 + \pi_2 \leqslant 1$ hold.

$s,h,\rho \models x = y$	$\Leftrightarrow$	$s(x) = s(y)$
$s,h,\rho \models \neg B$	$\Leftrightarrow$	$s,h,\rho \not\models B$
$s,h,\rho \models A \wedge B$	$\Leftrightarrow$	$s,h,\rho \models A$ and $s,h,\rho \models B$
$s,h,\rho \models \top$	$\Leftrightarrow$	always holds
$s,h,\rho \models \mathtt{emp}$	$\Leftrightarrow$	$dom(h) = \varnothing$
$s,h,\rho \models x \mapsto y$	$\Leftrightarrow$	$dom(h) = s(x)$ and
		$h(s(x)) = (s(y), 1)$
$s,h,\rho \models \mathtt{ls}(x,y)$	$\Leftrightarrow$	$s,h,\rho \models \mathtt{ls}^n(x,y)$ for some n
$s,h,\rho \models \mathtt{ls}^0(x,y)$	$\Leftrightarrow$	never holds
$s,h,\rho \models \mathtt{ls}^n(x,y)$	$\Leftrightarrow$	$s(x) = s(y) \wedge dom(h) = \varnothing$
(for $n > 0$)		or
		$\exists a \in \mathtt{Val}$
		$s[z := a], h, \rho \models x \mapsto z * \mathtt{ls}^n(z,y)$
$s,h,\rho \models B^\pi$	$\Leftrightarrow$	$\exists h'. h = \pi \bullet h'$
		and $s,h',\rho \models B$
$s,h,\rho \models \alpha$	$\Leftrightarrow$	$h = \rho(\alpha)$
$s,h,\rho \models @_\alpha B$	$\Leftrightarrow$	$s,\rho(\alpha),\rho \models B$
$s,h,\rho \models A * B$	$\Leftrightarrow$	$\exists h_1,h_2. h = h_1 \circ h_2$ and
		$s,h_1,\rho \models A$ and $s,h_2,\rho \models B$
$s,h,\rho \models A \circledast B$	$\Leftrightarrow$	$\exists h_1,h_2. h = h_1 \mathbin{\overline{\circ}} h_2$ and
		$s,h_1,\rho \models A$ and $s,h_2,\rho \models B$
$s,h,\rho \models A \mathrel{-\!\!*} B$	$\Leftrightarrow$	$\forall h'$. if $h \circ h'$ defined and
		$s,h',\rho \models A$ then $s,h \circ h',\rho \models B$
$s,h,\rho \models A \mathrel{-\!\!\circledast} B$	$\Leftrightarrow$	$\forall h'$. if $h \mathbin{\overline{\circ}} h'$ defined and
		$s,h',\rho \models A$ then $s,h \mathbin{\overline{\circ}} h',\rho \models B$

Figure1 The satisfaction relation for SL_{LP}

The weak composition $h_1 \mathbin{\overline{\circ}} h_2$ of two compatible permission heaps h_1 and h_2 is defined as defined as follows.

$$(h_1 \mathbin{\overline{\circ}} h_2)(l) = \begin{cases} (v, \pi_1 + \pi_2) & \text{if } h_1(l) = (v,\pi_1) \\ & \text{and } h_2(l) = (v,\pi_2) \\ h_1(l) & \text{if } l \notin dom(h_2) \\ h_2(l) & \text{if } l \notin dom(h_1). \end{cases}$$

If h_1 and h_2 are not *compatible* then $h_1 \mathbin{\overline{\circ}} h_2$ undefined.

Definition 2.2 (Semantics of SL_{LP})**.** *The satisfaction relation $s,h,\rho \models B$ for SL_{LP} is defined in Figure 1, where $\mathtt{ls}^n$ for $n \in \mathbb{N}$ are auxiliary predicates for inductive definition of $\mathtt{ls}$. $A \models B$ means that every permission heap models (s,h,ρ) satisfying the formula A also satisfies B. $A \equiv B$ holds iff $A \models B$ and $B \models A$.*

$x \mapsto y$ is points-to fact whose permission value is 1.0 as default. The predicate $\mathtt{ls}(x,y)$ expresses possibly-cyclic list segments. For example, $x \mapsto y * y \mapsto x \models \mathtt{ls}(x,x)$ holds.

2.3 Logical principles

Brotherston et al. [9] establish some logical entailments and equivalences of SL_{LP} w.r.t. separating conjunctions ($*$ and $\circledast$), fractional permissions and nominal labels.

Lemma 2.1 ([9])**.** *We have the following.*

$$A * B \equiv B * A \qquad A * \mathit{emp} \equiv A$$
$$A \circledast B \equiv B \circledast A \qquad A \circledast \mathit{emp} \equiv A$$
$$A \circledast (B \circledast C) \equiv (A \circledast B) \circledast C$$
$$A * (B * C) \equiv (A * B) * C$$
$$A * B \models A \circledast B$$

Lemma 2.2 (Permission division and combination [9])**.** *For all labels α and permission values π and σ, the following formulas hold.*

$$\alpha^\pi \circledast \alpha^\sigma \equiv \alpha^{\pi+\sigma} \qquad (+)$$
$$\alpha^{\pi+\sigma} \models \alpha^\pi \circledast \alpha^\sigma \qquad (\textsc{Split}\ \circledast)$$
$$\alpha^\pi \circledast \alpha^\sigma \models \alpha^{\pi+\sigma} \qquad (\textsc{Join}\ \circledast)$$

We use (Split $\circledast$) and (Join $\circledast$) to reason concurrent scene. For example, let us say that a shared variable is labeled by label α. Then, if two concurrent threads want to use the variable, then we express that each thread has $\alpha^{0.5}$. It means that each thread has read-only permission of the variable.

Lemma 2.3 ($\circledast$/$*$ distribution [9])**.** *For any permission heaps h_1,h_2,h_3 and h_4 s.t. $(h_1 \mathbin{\overline{\circ}} h_2) \circ (h_3 \mathbin{\overline{\circ}} h_4)$ is defined, then the following hold.*

$$(h_1 \mathbin{\overline{\circ}} h_2) \circ (h_3 \mathbin{\overline{\circ}} h_4) = (h_1 \circ h_3) \mathbin{\overline{\circ}} (h_2 \circ h_4)$$
$$(A \circledast B) * (C \circledast D) \equiv (A * C) \circledast (B * D)$$

3 Symbolic heaps with labels and permissions

We show our fragment SL_{LP}^{SH} restricting the formulas of SL_{LP} to the symbolic heaps. Here, SH stands for Symbolic Heap.

3.1 Symbolic heaps

Definition 3.1 (Spatial formulas)**.** *Spatial formu-*

las of SL_{LP}^{SH} are defined as follows.

$$S ::= x \mapsto y \mid \mathtt{ls}(x, y) \qquad \textit{spatial atoms}$$
$$\Sigma^- ::= \mathtt{emp} \mid S \mid \Sigma^- * \Sigma^- \qquad \textit{label-free}$$
$$\Sigma ::= \alpha^\pi \mid \Sigma^- \mid \Sigma * \Sigma \mid \Sigma \circledast \Sigma \qquad \textit{spatial formulas}$$

In SL_{LP}^{SH} , permission values less than 1 are allowed only with labels α^π, and permission values in spatial atoms are 1 as default. Σ^- contains neither labels nor $\circledast$.

Definition 3.2 (Pure formulas)**.** *Pure formulas of SL_{LP}^{SH} are defined as follows.*

$$P ::= x = y \mid x \neq y \mid @_\alpha \Sigma^- \qquad \textit{pure atoms}$$
$$\Pi ::= \top \mid P \mid \Pi \wedge \Pi \qquad \textit{pure formula}$$

We can notice that label α only denotes a label-free formula Σ^-.

Definition 3.3 (Symbolic heaps)**.** *Symbolic heaps of SL_{LP}^{SH} are defined as follows.*

$$\varphi := \Pi \,\text{¦}\, \Sigma$$

We suppose that for a symbolic heap $\Pi \,/\, \Sigma$, $@_\alpha \Sigma^-$ appears in Π exactly once for each label α in Σ. The semantics of $\Pi \,/\, \Sigma$ is the same as $\Pi \wedge \Sigma$.

For instance, $@_\alpha \Sigma_1 \wedge @_\alpha \Sigma_2 \,\text{¦}\, \alpha$ having two $@_\alpha$ formulas for label α is not allowed as a symbolic heap. $@_\alpha \Sigma_1 \,\text{¦}\, \alpha * \beta$ that does not have a $@_\beta$ formula for label β is not allowed.

Lemma 3.1 (Label Introduction)**.** *For any label-free spatial formula Σ^- and any label α, the following holds.*

$$\alpha \wedge \Sigma^- \equiv @_\alpha \Sigma^- \,/\, \alpha$$

Proof. Let s, h and ρ be given and we have

$s, h, \rho \models \alpha \wedge \Sigma^-$
$\Leftrightarrow$ $s, h, \rho \models \alpha$ and $s, h, \rho \models \Sigma^-$
$\Leftrightarrow$ $h = \rho(\alpha)$ and $s, h, \rho \models \Sigma^-$
$\Leftrightarrow$ $s, \rho(\alpha), \rho \models \Sigma^-$ and $h = \rho(\alpha)$
$\Leftrightarrow$ $s, h, \rho \models @_\alpha \Sigma^- \,\text{¦}\, \alpha$.

□

Lemma 3.2 (Label Elimination)**.** *For any pure formula Π, label-free spatial formula Σ^-, label α, the following holds.*

$$@_\alpha \Sigma^- \wedge \Pi \,/\, \alpha * \Sigma \models \Pi \,/\, \Sigma^- * \Sigma$$

Proof. Let s, h and ρ be given and we have

$s, h, \rho \models @_\alpha \Sigma^- \wedge \Pi \,\text{¦}\, \alpha * \Sigma$
$\Leftrightarrow$ $s, h, \rho \models @_\alpha \Sigma^-$ and $s, h, \rho \models \Pi$ and
$\exists h_1, h_2. h = h_1 \circ h_2$ and
$s, h_1, \rho \models \alpha$ and $s, h_2, \rho \models \Sigma$
$\Leftrightarrow$ $s, \rho(\alpha), \rho \models \Sigma^-$ and $s, h, \rho \models \Pi$ and
$\exists h_1, h_2. h = h_1 \circ h_2$ and
$h_1 = \rho(\alpha)$ and $s, h_2, \rho \models \Sigma$
$\Rightarrow$ $s, h, \rho \models \Pi$ and $\exists h_1, h_2. h = h_1 \circ h_2$ and
$s, h_1, \rho \models \Sigma^-$ and $s, h_2, \rho \models \Sigma$
$\Leftrightarrow$ $s, h, \rho \models \Pi \,\text{¦}\, \Sigma^- * \Sigma$

□

3.2 Proof rules

We introduce proof rules based on [9] in Figure 2, where $\mathtt{ModVars}(C)$ is the set of variables that program C may update them.

The original Frame rule in [9] is also applicable in labeled spatial formulas and spatial formulas within @ formulas in pure part in a symbolic heap. We will call these rules as (Frame Label) and (Frame @). Frame Label rule excludes the label α and $@_\alpha \Sigma_1$. Frame @ rule excludes Σ_1 in $@_\alpha(\Sigma_1 * \Sigma_2)$. The label of @ formula becomes a fresh label β because the labeled spatial formula is changed from $\Sigma_1 * \Sigma_2$ to Σ_2.

4 Examples of Concurrent program verification

This section, we prove two concurrent programs in our fragment. The first example is from [9][19]. We give the second one to show the expressiveness of our system in a scene of multiple and nested concurrent threads. In proof, we write captions for derivations of formulas by proof rule in red letters (e.g. (Frame Label)) and the ones by logical en-

$$\text{(Frame }*\text{)}(\boxtimes)\quad \frac{\{\Pi \mid \Sigma\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * \Sigma_1\}\ C\ \{\Pi' \mid \Sigma' * \Sigma_1\}}$$

$$\text{(Frame }\circledast\text{)}(\boxtimes)\quad \frac{\{\Pi \mid \Sigma\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma \circledast \Sigma_1\}\ C\ \{\Pi' \mid \Sigma' \circledast \Sigma_1\}}$$

$$\text{(Frame Label)}(\dagger,\boxtimes)\quad \frac{\{\Pi \mid \Sigma\}\ C\ \{\Pi' \mid \Sigma'\}}{\{@_\alpha\Sigma_1 \wedge \Pi \mid \alpha * \Sigma\}\ C\ \{@_\alpha\Sigma_1 \wedge \Pi' \mid \alpha * \Sigma'\}}$$

$$\text{(Frame @)}(\ddagger,\boxtimes)\quad \frac{\{@_\beta\Sigma_2 \wedge \Pi \mid \Sigma\}\ C\ \{@_\beta\Sigma_2 \wedge \Pi' \mid \Sigma'\}}{\{@_\alpha(\Sigma_1 * \Sigma_2) \wedge \Pi \mid \Sigma\}\ C\ \{@_\alpha(\Sigma_1 * \Sigma_2) \wedge \Pi' \mid \Sigma'\}}$$

$$\text{(Parallel)}(\oslash)\quad \frac{\{\Pi_1 \mid \Sigma_1\}\ C_1\ \{\Pi_1' \mid \Sigma_1'\} \quad \{\Pi_2 \mid \Sigma_2\}\ C_2\ \{\Pi_2' \mid \Sigma_2'\}}{\{\Pi_1 \wedge \Pi_2 \mid \Sigma_1 \circledast \Sigma_2\}\ C_1 \parallel C_2\ \{\Pi_1' \wedge \Pi_2' \mid \Sigma_1' \circledast \Sigma_2'\}}$$

$(\boxtimes)$: $\texttt{ModVars}(C) \cap FV(\Sigma_1) = \varnothing$

$(\dagger)$: α is fresh

$(\ddagger)$: α and β are fresh

$(\oslash)$: $\texttt{ModVars}(C_2) \cap (FV(\Pi_1 \mid \Sigma_1) \cup FV(\Pi_1' \mid \Sigma_1')) = \texttt{ModVars}(C_1) \cap (FV(\Pi_2 \mid \Sigma_2) \cup FV(\Pi_2' \mid \Sigma_2')) = \varnothing$

Figure2 Frame rules and Parallel rule

tailment in green letters (e.g. (Join $\circledast$)).

Our proofs employ the standard proof rules of *Concurrent Separation Logic* [8], and new ones in Figure 2.

4.1 Parallel read

Consider the program

$$C_1 = \texttt{foo (x, y) || foo (x, y)}.$$

Here, `foo (x, y)` only reads list segments $\texttt{ls}(x, y)$, so whose specification is $\{@_\alpha\texttt{ls}(x, y) \mid \alpha^\pi\}$ `foo(x,y)` $\{@_\alpha\texttt{ls}(x, y) \mid \alpha^\pi\}$ for any π. C_1 reads list segments $\texttt{ls}(x, y)$ concurrently by using `foo (x, y)` of each thread. The proof of $\{\top \mid \texttt{ls}(x, y)\}\ C_1\ \{\top \mid \texttt{ls}(x, y)\}$ is in Figure 3.

The proof uses the basic operations of SL_{LP}: *labeling* a formula with label α, *splitting* permission values of α and *joining* them again. We do labeling a formula $\top \mid \texttt{ls}(x, y)$ with α by (Label Intro). Then, we split permission values of α by (Split $\circledast$) to give read-only permission to two parallel threads executing `foo(x, y)` respectively. After the parallel call, the two copies $(@_\alpha\texttt{ls}(x, y) \mid \alpha^{0.5})$ are joined back together $(@_\alpha\texttt{ls}(x, y) \mid \alpha^{0.5} \circledast \alpha^{0.5})$ because they have the same label α. With (Join $\circledast$) rule,

$\{\top \mid \texttt{ls}(x, y)\}$

$\{@_\alpha\texttt{ls}(x, y) \mid \alpha\}$ (Label Intro) of α

$\{@_\alpha\texttt{ls}(x, y) \mid \alpha^{0.5} \circledast \alpha^{0.5}\}$ (Split $\circledast$) of α

$\{@_\alpha\texttt{ls}(x, y) \mid \alpha^{0.5}\}$	$\parallel$	$\{@_\alpha\texttt{ls}(x, y) \mid \alpha^{0.5}\}$ (Par)
`foo (x, y)`		`foo (x, y)`
$\{@_\alpha\texttt{ls}(x, y) \mid \alpha^{0.5}\}$		$\{@_\alpha\texttt{ls}(x, y) \mid \alpha^{0.5}\}$

$\{@_\alpha\texttt{ls}(x, y) \mid \alpha^{0.5} \circledast \alpha^{0.5}\}$ (Par)

$\{@_\alpha\texttt{ls}(x, y) \mid \alpha\}$ (Join $\circledast$) of α

$\{\top \mid \texttt{ls}(x, y)\}$ (Label Elimination) of α

Figure3 Proof of the program in Section 4.1

we can calculate $\alpha^{0.5} \circledast \alpha^{0.5}$ to $\alpha^{1.0}$. Finally, we eliminate label α in the formula by (Label Elimination) because we end proving the concurrent scene.

Main difference between labeling style of SL_{LP} $(\alpha \wedge \texttt{ls}(x, y))$ and one of SL_{LP}^{SH} $(@_\alpha\texttt{ls}(x, y) \mid \alpha)$ is that our system can maintain the form of symbolic heap after labeling.

4.2 Nested parallel read

Consider the program.

`foo (x,y) ||`

`foo (x,y); ( foo (a,b) || foo (a,b) )`

This program deals two different list segments $\texttt{ls}(x, y)$ and $\texttt{ls}(a, b)$. There are two main concurrent threads. One thread only reads $\texttt{ls}(x, y)$ by `foo (x,y)`. The other one first deals $\texttt{ls}(x, y)$ by `foo (x,y)` and processes $\texttt{ls}(a, b)$ with `foo (a,b) || foo (a,b)` sequentially. There is a proof of this program in Figure 4.

$\texttt{ls}(x, y)$ and $\texttt{ls}(a, b)$ are labeled with α and β, respectively. In the first parallel scene, α is split, and the right thread has $@_\beta\texttt{ls}(a, b)$ and β because left thread does not deal $\texttt{ls}(a, b)$. In the second parallel scene, $@_\alpha\texttt{ls}(x, y)$ and α are excluded by (Frame Label) because both nested concurrent threads do not use $\texttt{ls}(x, y)$. After two concurrent scenes are over, (Par) enables a confluence of α and β, i.e., $\alpha^{0.5} \circledast (\alpha^{0.5} * \beta)$. With Lemma 2.3 in [9] and Join $\circledast$, we can derive $\alpha * \beta$ from $\alpha^{0.5} \circledast (\alpha^{0.5} * \beta)$. This procedure of derivation is similar to *normalization* in Section 5. Finally, α and β is eliminated by (Label Elimination).

5 Decidable Entailment Checking by normalization

Let us say that there are two symbolic heaps φ and ψ. Entailment problem is checking $\varphi \models \psi$. Without permission values nor labels, it is *decidable* to solve the entailment problem for symbolic heap with list segments predicate [2][10]. Our approach to the entailment problem is to check entailment of normal forms. The normalization calculates fractional permission values of each label. Consequently, a normal form is $\circledast$-free and each label occurs in it at most once. By normalization, we can reduce our entailment checking to the one of [21][16] known as *decidable*.

5.1 Overall flow

For example, we want to check whether the entailment relation is valid or not. This example from Figure 4 is valid by Lemma 2.3.

$$@_\alpha\texttt{ls}(x, y) \land @_\beta\texttt{ls}(a, b) \mid (\alpha^{0.5} \circledast \alpha^{0.5}) * \beta \overset{?}{\vdash}$$
$$@_\alpha\texttt{ls}(x, y) \land @_\beta\texttt{ls}(a, b) \mid \alpha^{0.5} \circledast (\alpha^{0.5} * \beta)$$

First, we derive normal forms by *normalization*.

$$@_\alpha\texttt{ls}(x, y) \land @_\beta\texttt{ls}(a, b) \mid \alpha * \beta \overset{?}{\vdash}$$
$$@_\alpha\texttt{ls}(x, y) \land @_\beta\texttt{ls}(a, b) \mid \alpha * \beta$$

Second, we eliminate all labels by our new entailment rules (Label Equality Check) and (Left Label elimination) in Figure 6. For the above example, we have

$$\top \mid \texttt{ls}(x, y) * \texttt{ls}(a, b) \overset{?}{\vdash} \top \mid \texttt{ls}(x, y) * \texttt{ls}(a, b)$$

by applying Label Equality Check.

Finally, the symbolic heaps become label-free, permission-free, and $\circledast$-free. That is, we can check by existing entailment checking algorithm of [16][21].

5.2 Normalization

The normalization calculates permission values of each same label until a symbolic heap becomes a normal form.

Definition 5.1 (Normal forms)**.** *The normal spatial formula is defined as*

$$\Sigma_{nf} ::= \alpha^\pi \mid \Sigma^- \mid \Sigma_{nf} * \Sigma_{nf}$$

such that each label occurs at most once. A symbolic heap $\Pi \mid \Sigma_{nf}$ *such that* Σ_{nf} *is a normal spatial formula is called a normal form.*

We define the function $label_{sh}(\Pi \mid \Sigma)$ collecting labels from symbolic heaps.

Definition 5.2. $label_s(\Sigma) \in \mathcal{P}(\texttt{Label})$ *is defined as follows.*

- $label_s(\alpha^\pi) := \{\alpha\}$
- $label_s(\Sigma_1 * \Sigma_2) := label_s(\Sigma_1) \cup label_s(\Sigma_2)$
- $label_s(\Sigma_1 \circledast \Sigma_2) := label_s(\Sigma_1) \cup label_s(\Sigma_2)$
- $label_s(\Sigma) := \varnothing$ *(if there is not a label in* Σ*)*

$label_p(\Pi) \in \mathcal{P}(\texttt{Label})$ *is defined as follows.*

- $label_p(@_\alpha\Sigma) := label_s(\Sigma)$

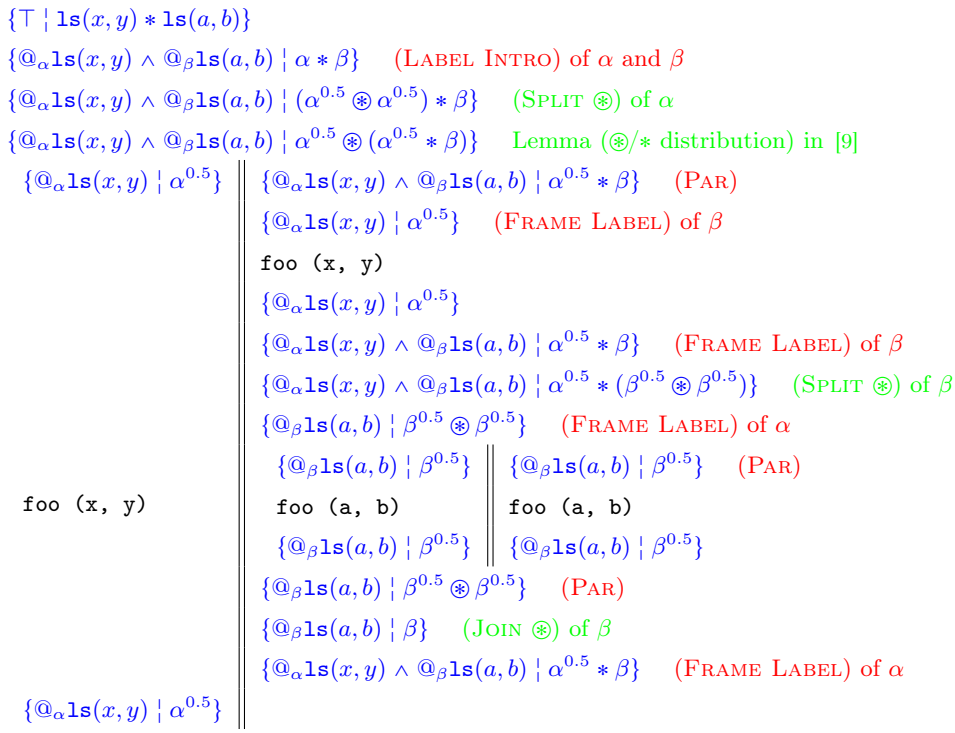

Figure4 Proof of the program in Section 4. 2

- $label_p(\Pi_1 \wedge \Pi_2) := label_p(\Pi_1) \cup label_p(\Pi_2)$
- $label_p(\Pi) := \varnothing$

  (if there is not a @ formula in Π*)*

$label_{sh}(\Pi \mid \Sigma) \in \mathcal{P}(\texttt{Label})$ *is defined as follows.*

- $label_{sh}(\Pi \mid \Sigma) := label_p(\Pi) \cup label_s(\Sigma)$

Definition 5.3 (Normalization)**.** *The normalization function (normalizaton) on symbolic heaps is defined in Figure 5.*

Note that $\Pi \mid \alpha^{\pi} * \alpha^{\sigma}$ entails that the heap denoted by α is empty. $n_E(\Sigma)$ is the set of labels that must denote the empty heap if $\Pi \mid \Sigma$ holds.

We show that symbolic heaps before and after *normalization* are logically equivalent (Proposition 5.1) and. *normalization* always generates normal form (Proposition 5.2). To prove Proposition 5.1, there are several lemmas.

Lemma 5.1. *If* $s, h, \rho \models \Pi \mid \Sigma$ *and* $\alpha \in n_E(\Sigma)$*, then* $dom(\rho(\alpha)) = \varnothing$*.*

Proof. We use induction on Σ.

1. For Σ^{-}, $n_E(\Sigma^{-}) = \varnothing$
2. For α, $n_E(\alpha) = \varnothing$
3. For $\Sigma_1 \circledast \Sigma_2$, $n_E(\Sigma_1 \circledast \Sigma_2) = n_E(\Sigma_1) \cup n_E(\Sigma_2)$
4. For $\Sigma_1 * \Sigma_2$, Let us say that $\alpha \in (label_s(\Sigma_1) \cap label_s(\Sigma_2)) \cup n_E(\Sigma_1) \cup n_E(\Sigma_2)$.
   - $\alpha \in label_s(\Sigma_1) \cap label_s(\Sigma_2)$

     We will prove this case by showing contradiction.

     $s, h, \rho \models \Pi \mid \Sigma_1 * \Sigma_2 \overset{def}{\iff} \exists h_1, h_2. h = h_1 \circ h_2$ (i.e., $dom(h_1) \cap dom(h_2) = \varnothing$) and $s, h_1, \rho \models \Pi \mid \Sigma_1$ and $s, h_2, \rho \models \Pi \mid \Sigma_2$

     Because $\alpha \in label_s(\Sigma_1) \cap label_s(\Sigma_2)$,

1. $n_s(\Sigma)$:
   - $n_s(\Sigma^-) = \Sigma^-$
   - $n_s(\alpha^\pi) = \alpha^\pi$
   - $n_s(\Sigma_1 * \Sigma_2) =$
     $\ast_{i=1}^n {\alpha_i}^{\pi_i+\sigma_i} * {\Sigma_1}' * {\Sigma_2}'$
     - $n_s(\Sigma_1) = \ast_{i=1}^n \alpha_i{}^{\pi_i} * {\Sigma_1}'$
     - $n_s(\Sigma_2) = \ast_{i=1}^n \alpha_i{}^{\sigma_i} * {\Sigma_2}'$
     - $\alpha_{1\leqslant i\leqslant n} \in label_s(\Sigma_1) \cap label_s(\Sigma_2)$
   - $n_s(\Sigma_1 \circledast \Sigma_2) = n_s(\Sigma_1 * \Sigma_2)$
2. $n_E(\Sigma) \in \mathcal{P}(labels)$:
   - $n_E(\Sigma_1 * \Sigma_2) = (label_s(\Sigma_1) \cap label_s(\Sigma_2))$
     $\cup\, n_E(\Sigma_1) \cup n_E(\Sigma_2)$
   - $n_E(\Sigma_1 \circledast \Sigma_2) = n_E(\Sigma_1) \cup n_E(\Sigma_2)$
   - $n_E(\Sigma^-) = n_E(\alpha^\pi) = \varnothing$
3. $n_p(\Pi, L)$ for $L \in \mathcal{P}(\mathtt{Label})$:
   - $n_p(\Pi, L) = \Pi$ (if $label_p(\Pi) \cap L \neq \varnothing$)
   - $n_p(@_\alpha \ast_{i=1}^n \mathtt{ls}(s_i, t_i), L) =$
     $@_\alpha \mathtt{emp} \wedge \bigwedge_{i=1}^n s_i = t_i$ (if $\alpha \in L$)
   - $n_p(@_\alpha \Sigma^-, L) = \mathtt{nil} \neq \mathtt{nil} \wedge @_\alpha \mathtt{emp}$
     (if $\alpha \in L$, and $\Sigma^- \neq \ast_{i=1}^n \mathtt{ls}(s_i, t_i)$)
   - $n_p(\Pi_1 \wedge \Pi_2, L) = n_p(\Pi_1, L) \wedge n_p(\Pi_2, L)$
4. *normalization* on symbolic heaps:
   - $normalization(\Pi \mid \Sigma) =$
     $n_p(\Pi, n_E(\Sigma)) \mid n_s(\Sigma)$.

Figure5 Normalization

$\rho(\alpha) \subseteq dom(h_1)$ and $\rho(\alpha) \subseteq dom(h_2)$ hold. If $dom(\rho(\alpha)) \neq \varnothing$ then $dom(h_1) \cap dom(h_2) \neq \varnothing$, this is contradiction.

- $\alpha \notin label_s(\Sigma_1) \cap label_s(\Sigma_2)$
  - $\alpha \in n_E(\Sigma_1)$
    $s, h, \rho \models \Pi \mid \Sigma_1 * \Sigma_2 \overset{def}{\iff} \exists h_1, h_2. h = h_1 \circ h_2$ (i.e., $dom(h_1) \cap dom(h_2) = \varnothing$) and $s, h_1, \rho \models \Pi \mid \Sigma_1$ and $s, h_2, \rho \models \Pi \mid \Sigma_2$. By induction hypothesis, $dom(\rho(\alpha)) = \varnothing$.
  - $\alpha \in n_E(\Sigma_2)$
    The proof is the same as the above.

□

Lemma 5.2. *If* $s, h, \rho \models @_\alpha \ast_{i=1}^n \mathtt{ls}(s_i, t_i) \wedge \Pi \,/\, \Sigma$ *and* $dom(\rho(\alpha)) = \varnothing$, *then for all* i, $s(s_i) = s(t_i)$.

Proof. $s, h, \rho \models @_\alpha \ast_{i=1}^n \mathtt{ls}(s_i, t_i)$
$\Rightarrow s, \rho(\alpha), \rho \models \ast_{i=1}^n \mathtt{ls}(s_i, t_i)$
$\Rightarrow s, \rho(\alpha), \rho \models \mathtt{ls}(s_i, t_i)$ // for all i
$\Rightarrow s(s_i) = s(t_i)$ // for all i □

Lemma 5.3. *If* $\alpha \in n_E(\Sigma)$ *and* $\Sigma_\mapsto$ *contains* $\mapsto$, *then* $s, h, \rho \not\models @_\alpha \Sigma_\mapsto \wedge \Pi \,/\, \Sigma$.

Proof. We will prove this proposition by showing contradiction. By Proposition 5.1, if $\alpha \in n_E(\Sigma)$, then $dom(\rho(\alpha)) = \varnothing$. Let us say that

$$s, h, \rho \models @_\alpha x \mapsto y \iff s, \rho(\alpha), \rho \models x \mapsto y$$

This is contradiction because $s, h, \rho \models x \mapsto y$ does not hold if $dom(h) = \varnothing$. □

Lemma 5.4.

If $s, h, \rho \models \Sigma_1 \circledast \Sigma_2$ and $label_s(\Sigma_1) \cap label_s(\Sigma_2) = \varnothing$, then $s, h, \rho \models \Sigma_1 * \Sigma_2$.

Proof. $s, h, \rho \models \Pi \mid \Sigma_1 \circledast \Sigma_2 \overset{def}{\iff} \exists h_1, h_2. h = h_1 \,\overline{\circ}\, h_2$ and $s, h_1, \rho \models \Pi \mid \Sigma_1$ and $s, h_2, \rho \models \Pi \mid \Sigma_2$

There is not a label α s.t. $\alpha \in dom(h_1) \cap dom(h_2)$ and for all different labels β and γ, $dom(\rho(\beta)) \cap dom(\rho(\gamma)) = \varnothing$. Therefore, $h_1 \,\overline{\circ}\, h_2 = h_1 \circ h_2$ and $s, h, \rho \models \Pi \mid \Sigma_1 * \Sigma_2$ holds. □

Proposition 5.1. $\Pi \,/\, \Sigma \equiv normalization(\Pi \,/\, \Sigma)$

Proof. We will prove this proposition by using induction on Σ.

1. $n_s(\Sigma^-) = \Sigma^-$, so this case is obvious.
2. $n_s(\alpha^\pi) = \alpha^\pi$, so this case is obvious.
3. For $n_s(\Sigma_1 * \Sigma_2) = \ast_{i=1}^n \alpha_i{}^{\pi_i+\sigma_i} * {\Sigma_1}' * {\Sigma_2}'$, we will use induction. By induction hypothesis, (a) and (b) satisfies Proposition 5.1.
   (a) $n_s(\Sigma_1) = \ast_{i=1}^n \alpha_i{}^{\pi_i} * {\Sigma_1}'$
   (b) $n_s(\Sigma_2) = \ast_{i=1}^n \alpha_i{}^{\sigma_i} * {\Sigma_2}'$

$(\alpha_{1\leqslant i\leqslant n} \in label_s(\Sigma_1) \cap label_s(\Sigma_2))$

$s,h,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \Sigma_1 * \Sigma_2$

$\Longleftrightarrow \exists h_1, h_2. h = h_1 \circ h_2$ and

$s,h_1,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \Sigma_1$ and

$s,h_2,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \Sigma_2$

$\Longleftrightarrow s,h_1,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \circledast_{i=1}^{n} {\alpha_i}^{\pi_i} * \Sigma_1'$ and

$s,h_2,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \circledast_{i=1}^{n} {\alpha_i}^{\pi_i} * \Sigma_2'$

(By the induction hypothesis)

$\Longleftrightarrow s,h,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \circledast_{i=1}^{n} {\alpha_i}^{\pi_i+\sigma_i} * \Sigma_1' * \Sigma_2'$

(Since, for all $1 \leqslant i \leqslant n$, $dom(\rho(\alpha_i)) = \varnothing$)

4. For $n_s(\Sigma_1 \circledast \Sigma_2) = \circledast_{i=1}^{n} {\alpha_i}^{\pi_i+\sigma_i} * \Sigma_1' * \Sigma_2'$, we will use induction. By induction hypothesis, (a) and (b) satisfies Proposition 5.1.

   (a) $n_s(\Sigma_1) = \circledast_{i=1}^{n} {\alpha_i}^{\pi_i} * \Sigma_1'$

   (b) $n_s(\Sigma_2) = \circledast_{i=1}^{n} {\alpha_i}^{\sigma_i} * \Sigma_2'$

   $(\alpha_{1\leqslant i\leqslant n} \in label_s(\Sigma_1) \cap label_s(\Sigma_2))$

$s,h,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \Sigma_1 \circledast \Sigma_2$

$\Longleftrightarrow \exists h_1, h_2. h = h_1 \bar{\circ} h_2$ and

$s,h_1,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \Sigma_1$ and

$s,h_2,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \Sigma_2$

$\Longleftrightarrow s,h_1,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \circledast_{i=1}^{n} {\alpha_i}^{\pi_i} * \Sigma_1'$ and

$s,h_2,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \circledast_{i=1}^{n} {\alpha_i}^{\pi_i} * \Sigma_2'$

(By the induction hypothesis)

$\Longleftrightarrow s,h,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \circledast_{i=1}^{n} {\alpha_i}^{\pi_i+\sigma_i} * (\Sigma_1' \circledast \Sigma_2')$

(By Lemma 2.2 and Lemma 2.3)

$\Longleftrightarrow s,h,\rho \models \bigwedge_{i=1}^{n} @_{\alpha_i}\mathtt{emp} \wedge \Pi \mid \circledast_{i=1}^{n} {\alpha_i}^{\pi_i+\sigma_i} * (\Sigma_1' * \Sigma_2')$

(By Lemma 5.4)

□

Proposition 5.2. *Normalization derives normal form.*

Proof. We will prove that $n_s(\Sigma)$ is a normal form by using induction on Σ.

1. $n_s(\Sigma^-) = \Sigma^-$, and Σ^- is normal form.
2. $n_s(\alpha^\pi) = \alpha^\pi$, and α^π is the normal form.
3. For $n_s(\Sigma_1 * \Sigma_2)$, we will use induction. By the induction hypothesis, (a) and (b) are normal forms.

   (a) $n_s(\Sigma_1) = \circledast_{i=1}^{n} {\alpha_i}^{\pi_i} * \Sigma_1'$

   (b) $n_s(\Sigma_2) = \circledast_{i=1}^{n} {\alpha_i}^{\sigma_i} * \Sigma_2'$

   $(\alpha_{1\leqslant i\leqslant n} \in label_s(\Sigma_1) \cap label_s(\Sigma_2))$

   By the induction hypothesis, Σ_1' and Σ_2' are normal forms, so Σ_1' and Σ_2' are also normal forms. Consequently, $n_s(\Sigma_1 * \Sigma_2) = \circledast_{i=1}^{n} {\alpha_i}^{\pi_i+\sigma_i} * \Sigma_1' * \Sigma_2'$ is normal form. That is, each label α_i appears at most once and $\circledast$ is eliminated.
4. For $n_s(\Sigma_1 \circledast \Sigma_2)$, $n_s(\Sigma_1 \circledast \Sigma_2) = n_s(\Sigma_1 * \Sigma_2)$ and $n_s(\Sigma_1 * \Sigma_2)$ is normal form, so $n_s(\Sigma_1 \circledast \Sigma_2)$ is also normal form.

□

5.3 Decidable entailment check of normal form

After normalization of symbolic heaps, we use entailment check rules in Figure 6 to exclude labels with fractional permissions and corresponding @ formulas. We apply these rules repeatedly, then symbolic heaps become *label-free* and *permission-free*. Therefore, it is possible to reduce our entailment checking problem to existing one of [21][16] known as *decidable*.

We show that the rules in Figure 6 are locally sound and complete, that is, the upper entailment is valid if and only if the lower one is valid.

Lemma 5.5. *The rules in Figure 6 are locally sound and complete.*

Then, the entailment check algorithm for normal forms is in Figure 7. This procedure terminates since the number of labels in LHS decreases by applying each rule in Figure 6.

For line 1 in Figure 7, the satisfiability problem of normal form in SL_{LP}^{SH} can be solved by reducing the satisfiability of symbolic heaps without permission values in [10]. For normal form $@_\alpha \Sigma^- \wedge \Pi \mid \alpha^\pi * \Sigma$,

$$\frac{\Pi \mid \Sigma_1 * \Sigma \vdash \Pi' \mid \Sigma_1{}' * \Sigma'}{@_\alpha \Sigma_1 \wedge \Pi \mid \alpha^\pi * \Sigma \vdash @_\alpha \Sigma_1{}' \wedge \Pi' \mid \alpha^\pi * \Sigma'}\ (\text{Label Equality Check})(\dagger)$$

$$\frac{\Pi \mid \Sigma_1 * \Sigma \vdash \Pi' \mid \Sigma'}{@_\alpha \Sigma_1 \wedge \Pi \mid \alpha * \Sigma \vdash \Pi' \mid \Sigma'}\ (\text{Left Label Elimination})$$

$$\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{@_\alpha \texttt{emp} \wedge \Pi \mid \alpha^\pi * \Sigma \vdash \Pi' \mid \Sigma'}\ (\text{Empty Label Elimination})(\dagger)$$

($\dagger$) : α is fresh

Figure6 Proof rules for Entailments of normal form

Input: $\varphi \vdash \psi$ (φ and ψ are normal forms)
Output: *Valid* or *Invalid*

1: **if** φ is satisfiable **then**
2: **if** $label_{sh}(\psi) - label_{sh}(\varphi) \neq \varnothing$ **then**
3: **return** *Invalid*
4: **else if** $\alpha_i \in label_{sh}(\varphi) \cap label_{sh}(\psi)$ **then**
5: **if** permission values of α_i in φ and ψ are different **then**
6: **return** *Invalid* (Since, applying Label Equality Check is impossible)
7: **else apply** Label Equality Check for each α_i and **do** *EC* recursively
8: **end if**
9: **else if** $\alpha_i \in label_{sh}(\varphi) - label_{sh}(\psi)$ **then**
10: **if** $@_{\alpha_i}\Sigma^-$ (Σ^- is not `emp`) and permission value of α_i is less than 1 **then**
11: **return** *Invalid*
12: **else apply** Left Label Elimination or Empty Label Elimination
13: **end if**
14: **else apply** existing entailment checker (Since, input symbolic heaps are label-free)
15: **end if**
16: **else return** *Valid*
17: **end if**

Figure7 Entailment check algorithm : *EC*

we have
$@_\alpha\Sigma^- \wedge \Pi \mid \alpha^\pi * \Sigma$ is satisfiable
$\Longleftrightarrow$ $@_\alpha\Sigma^- \wedge \Pi \mid \alpha * \Sigma$ is satisfiable
(by forgetting permission values)
$\Longleftrightarrow$ $\Pi \mid \Sigma^- * \Sigma$: *satisfiable*
(by Label Elimination).

For lines 3, 6, and 11, we can see that any normal form entailment that contains a label and that no rule can be applied is invalid.

Lemma 5.6. *Suppose that ϕ and ψ are normal forms, and ϕ is satisfiable and contains a label. If $\phi \vdash \psi$ cannot be a lower entailment of either of the rules in Figure 6, then $\phi \vdash \psi$ is invalid.*

If all of the labels are eliminated by applying the rules in Figure 6, the resulting entailment is label-free. Hence, we can use the existing entailment checking algorithm (line 14).

In combination with normalization, we prove the decidability of the entailment checking in SL^{SH}_{LP} .

Theorem 5.1. *Entailment checking in SL_{LP}^{SH} is decidable.*

6 Conclusion

We propose the subsystem of the concurrent separation logic with fractional permissions [9] by restricting the formulas to symbolic heaps, which are suitably tractable fragments for automating the verification. Also, our result extends the scope of the decidable entailment checking system of Berdine et al. [4][21][16] to restricted concurrent separation logic with fractional permissions and labels. That is, the main contribution of this paper is dealing widely studied topics *symbolic heap* and *entailment checking* within the restricted subsystem of SL_{LP}.

Unlike Brotherston et al. [9], SL_{LP}^{SH} does not have separating implication ($-\!\!*$). Because we found out that including $-\!\!*$ in our syntax makes our logic undecidable. Therefore, to give up expressing $-\!\!*$ was for the cost of decidable entailment check in our current research. Unfortunately, without $-\!\!*$, we cannot describe a complicated scene of a concurrent program such that a part of the variable gets writable permission, so the value of this variable is updated, and other concurrent threads also use the variable.

Consequently, an obvious way to proceed is relaxing SL_{LP}^{SH} to cover a broader concurrent program. In our opinion, it is inevitable to include $-\!\!*$ in our syntax for this. We are willing to find another subsystem with $-\!\!*$ keeping our achievements in the current work.

An even greater challenge is solving *shape analysis* problem [14] based on our logic. This problem is discovering shapes of the heap structure during a program's execution as a form of pointer analysis. For example, w.r.t. $\mathtt{ls}(x, y)$, this checks aliasing of variable x and y and furthermore, deeper properties of the heap (e.g., is $\mathtt{ls}(x, y)$ acyclic?). This problem theoretically rests on *bi-abduction* discovering best fit solutions for applications of the FRAME rule [11][12]. In the current work's setting, our further work is deciding how applications of PARALLEL rule divide different labels between threads.

We hope this paper stimulates interest in symbolic heaps and their entailment checking of the concurrent logic system SL_{LP} of Brotherston et al. [9].

References

[1] Antonopoulos, T., Gorogiannis, N., Haase, C., Kanovich, M., and Ouaknine, J.: Foundations for decision problems in separation logic with general inductive predicates, *International Conference on Foundations of Software Science and Computation Structures*, Springer, 2014, pp. 411–425.

[2] Berdine, J., Calcagno, C., and O ' hearn, P. W.: A decidable fragment of separation logic, *International Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer, 2004, pp. 97–109.

[3] Berdine, J., Calcagno, C., and O ' hearn, P. W.: Smallfoot: Modular automatic assertion checking with separation logic, *International Symposium on Formal Methods for Components and Objects*, Springer, 2005, pp. 115–137.

[4] Berdine, J., Calcagno, C., and O ' hearn, P. W.: Symbolic execution with separation logic, *Asian Symposium on Programming Languages and Systems*, Springer, 2005, pp. 52–68.

[5] Blackburn, P.: Representation, reasoning, and relational structures: a hybrid logic manifesto, *Logic Journal of the IGPL*, Vol. 8, No. 3(2000), pp. 339–365.

[6] Bornat, R., Calcagno, C., O'Hearn, P., and Parkinson, M.: Permission accounting in separation logic, *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005, pp. 259–270.

[7] Boyland, J.: Checking interference with fractional permissions, *International Static Analysis Symposium*, Springer, 2003, pp. 55–72.

[8] Brookes, S.: A semantics for concurrent separation logic, *Theoretical Computer Science*, Vol. 375, No. 1-3(2007), pp. 227–270.

[9] Brotherston, J., Costa, D., Hobor, A., and Wickerson, J.: Reasoning over Permissions Regions in Concurrent Separation Logic, *International Conference on Computer Aided Verification*, Springer, 2020, pp. 203–224.

[10] Brotherston, J., Fuhs, C., Pérez, J. A. N., and Gorogiannis, N.: A decision procedure for satisfiability in separation logic with inductive predicates, *Proceedings of the Joint Meeting of the Twenty-*

Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), 2014, pp. 1–10.

[11] Brotherston, J., Gorogiannis, N., and Kanovich, M.: Biabduction (and related problems) in array separation logic, *International Conference on Automated Deduction*, Springer, 2017, pp. 472–490.

[12] Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H.: Compositional shape analysis by means of bi-abduction, *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009, pp. 289–300.

[13] Calcagno, C., Distefano, D., and O ' Hearn, P.: Open-sourcing Facebook Infer: Identify bugs before you ship, *code. facebook. com blog post*, Vol. 11(2015).

[14] Calcagno, C., O'Hearn, P. W., and Yang, H.: Local action and abstract separation logic, *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, IEEE, 2007, pp. 366–378.

[15] Distefano, D., O ' hearn, P. W., and Yang, H.: A local shape analysis based on separation logic, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2006, pp. 287–302.

[16] Iosif, R., Rogalewicz, A., and Simacek, J.: The tree width of separation logic with recursive definitions, *International Conference on Automated Deduction*, Springer, 2013, pp. 21–38.

[17] Jacobs, B., Smans, J., and Piessens, F.: VeriFast: Imperative programs as proofs, *VSTTE workshop on Tools & Experiments*, 2010.

[18] Katelaan, J., Matheja, C., and Zuleger, F.: Effective entailment checking for separation logic with inductive definitions, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2019, pp. 319–336.

[19] Le, X.-B. and Hobor, A.: Logical reasoning for disjoint permissions, *European Symposium on Programming*, Springer, 2018, pp. 385–414.

[20] Reynolds, J. C.: Separation logic: A logic for shared mutable data structures, *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, IEEE, 2002, pp. 55–74.

[21] Tatsuta, M., Nakazawa, K., and Kimura, D.: Completeness of cyclic proofs for symbolic heaps with inductive definitions, *Asian Symposium on Programming Languages and Systems*, Springer, 2019, pp. 367–387.