# Reasoning over Permissions Regions in Concurrent Separation Logic

James Brotherston[1], Diana Costa[1], Aquinas Hobor[2], and John Wickerson[3]

[1] University College London, UK
[2] National University of Singapore
[3] Imperial College London, UK

**Abstract.** We propose an extension of separation logic with fractional permissions, aimed at reasoning about concurrent programs that share arbitrary *regions* or data structures in memory. In existing formalisms, such reasoning typically either fails or is subject to stringent side conditions on formulas (notably *precision*) that significantly impair automation. We suggest two formal syntactic additions that collectively remove the need for such side conditions: first, the use of both "weak" and "strong" forms of separating conjunction, and second, the use of nominal labels from hybrid logic. We contend that our suggested alterations bring formal reasoning with fractional permissions in separation logic considerably closer to common pen-and-paper intuition, while imposing only a modest bureaucratic overhead.

**Keywords:** Separation logic, permissions, concurrency, verification.

## 1 Introduction

*Concurrent separation logic* (CSL) is a version of separation logic designed to enable compositional reasoning about concurrent programs that manipulate memory possibly shared between threads [6, 26]. Like standard separation logic [28], CSL is based on *Hoare triples* $\{A\}\,C\,\{B\}$, where $C$ is a program and $A$ and $B$ are formulas (called the *precondition* and *postcondition* of the code respectively). The heart of the formalism is the following *concurrency rule*:

$$\frac{\{A_1\}\,C_1\,\{B_1\} \quad \{A_2\}\,C_2\,\{B_2\}}{\{A_1 \circledast A_2\}\,C_1 \,||\, C_2\,\{B_1 \circledast B_2\}}$$

where $\circledast$ is a so-called *separating conjunction*. This rule says that if two threads $C_1$ and $C_2$ are run on spatially separated resources $A_1 \circledast A_2$ then the result will be the spatially separated result, $B_1 \circledast B_2$, of running the two threads individually.

However, since many or perhaps even most interesting concurrent programs do share some resources, $\circledast$ typically does not denote strict disjoint separation of memories, as it does in standard separation logic (where it is usually written as $*$). Instead, it usually denotes a weaker sort of "separation" designed to ensure

that the two threads at least cannot interfere with each others' data. This gives rise to the idea of *fractional permissions*, which allow us to divide writeable memory into multiple read-only copies by adding a permission value to each location in heap memory. In the usual model, due to Boyland [5], permissions are rational numbers in the half-open interval $(0, 1]$, with 1 denoting the write permission, and values in $(0, 1)$ denoting read-only permissions. We write the formula $A^\pi$, where $\pi$ is a permission, to denote a "$\pi$ share" of the formula $A$. For example, $(x \mapsto a)^{0.5}$ (typically written as $x \overset{0.5}{\mapsto} a$ for convenience) denotes a "half share" of a single heap cell, with address $x$ and value $a$. The separating conjunction $A \circledast B$ then denotes heaps realising $A$ and $B$ that are "compatible", rather than disjoint: where the heaps overlap, they must agree on the data value, and one adds the permissions at the overlapping locations [4]. E.g., at the logical level, we have the entailment:

$$x \overset{0.5}{\mapsto} a \circledast x \overset{0.5}{\mapsto} b \models a = b \wedge x \mapsto a \ . \tag{1}$$

Happily, the concurrency rule of CSL is still sound in this setting (see e.g. [29]).

However, the use of this weaker notion of separation $\circledast$ causes complications for formal reasoning in separation logic, especially if one wishes to reason over arbitrary regions of memory rather than individual pointers. There are two particular difficulties, as identified by Le and Hobor [24]. The first is that, since $\circledast$ denotes possibly-overlapping memories, one loses the main useful feature of separation logic: its nonambiguity about separation, which means that desirable entailments such as $A^{0.5} \circledast B^{0.5} \models (A \circledast B)^{0.5}$ turn out to be false. E.g.:

$$x \overset{0.5}{\mapsto} a \circledast y \overset{0.5}{\mapsto} b \not\models (x \mapsto a \circledast y \mapsto b)^{0.5} \ .$$

Here, the two "half-pointers" on the LHS might be aliased ($x = y$ and $a = b$), meaning they are two halves of the same pointer, whereas on the RHS they must be non-aliased (because we cannot combine two "whole" pointers). This ambiguity becomes quite annoying when one adds arbitrary predicate symbols to the logic, e.g. to support inductively defined data structures.

The second difficulty is that although recombining single pointers is straightforward, as indicated by equation (1), recombining the shares of arbitrary formulae is challenging. E.g., $A^{0.5} \circledast A^{0.5} \not\models A$, as shown by the counterexample

$$(x \mapsto 1 \vee y \mapsto 2)^{0.5} \circledast (x \mapsto 1 \vee y \mapsto 2)^{0.5} \not\models x \mapsto 1 \vee y \mapsto 2 \ .$$

The LHS can be satisfied by a heap with a 0.5-share of $x$ and a 0.5-share of $y$, whereas the RHS requires a full (1) share of either $x$ or $y$.

Le et al. [24] address these problems by a combination of the use of *tree shares* (essentially Boolean binary trees) rather than rational numbers as permissions, and semantic restrictions on when the above sorts of permissions reasoning can be applied. For example, recombining permissions ($A^{0.5} \circledast A^{0.5} \models A$) is permitted only when the formula is *precise* in the usual separation logic sense (cf. [28]). The chief drawback with this approach is the need to repeatedly check these side

conditions on formulas when reasoning, as well as that said reasoning cannot be performed on imprecise formulas.

Instead, we propose to resolve these difficulties by a different, two-pronged extension to the syntax of the logic. First, we propose that the usual "strong" separating conjunction $*$, which enforces the strict disjointness of memory, *should be retained* in the formalism in addition to the weaker $\circledast$. The stronger $*$ supports entailments such as $A^{0.5} * B^{0.5} \models (A * B)^{0.5}$, which does not hold when $\circledast$ is used instead. Second, we introduce *nominal labels* from hybrid logic (cf. [3, 10]) to remember that two copies of a formula have the same origin. We write a nominal $\alpha$ to denote a unique heap, in which case entailments such as $(\alpha \wedge A)^{0.5} \circledast (\alpha \wedge A)^{0.5} \models \alpha \wedge A$ become valid. We remark that labels have been adopted for similar "tracking" purposes in several other separation logic proof systems [10, 21, 23, 25].

The remainder of this paper aims to demonstrate that our proposed extensions are (i) weakly *necessary*, in that expected reasoning patterns fail under the usual formalism, (ii) *correct*, in that they recover the desired logical principles, and (iii) *sufficient* to verify typical concurrent programming patterns that use sharing. Section 2 gives some simple examples that motivate our extensions. Section 3 then formally introduces the syntax and semantics of our extended formalism. In Section 4 we show that our logic obeys the logical principles that enable us to reason smoothly with fractional permissions over arbitrary formulas, and in Section 5 we give some longer worked examples. Finally, in Section 6 we conclude and discuss directions for future work.

## 2 Motivating examples

In this section, we aim to motivate our extensions to separation logic with permissions by showing, firstly, how the failures of the logical principles described in the introduction actually arise in program verification examples and, secondly, how these failures are remedied by our proposed changes.

The overall context of our work is reasoning about concurrent programs that share some data structure or region in memory, which can be described as a formula in the assertion language. If $A$ is such a formula then we write $A^\pi$ to denote a "$\pi$ share" of the formula $A$, meaning informally that all of the pointers in the heap memory satisfying $A$ are owned with share $\pi$. The main question then becomes how this notion interacts with the separating conjunction $\circledast$. There are two key desirable logical equivalences:

$$(A \circledast B)^\pi \;\equiv\; A^\pi \circledast B^\pi \tag{I}$$

$$A^{\pi \oplus \sigma} \;\equiv\; A^\pi \circledast A^\sigma \tag{II}$$

Equivalence (I) describes distributing a fractional share over a separating conjunction, whereas equivalence (II) describes combining two pieces of a previously split resource. Both equivalences are true in the $\models$ direction but, as we have seen in the Introduction, false in the $\dashv\models$ one. Generally speaking, $\circledast$ is like Humpty Dumpty: easy to break apart, but not so easy to put back together again.

The key to understanding the difficulty is the following equivalence:

$$x \overset{\pi}{\mapsto} a \circledast y \overset{\sigma}{\mapsto} b \;\equiv\; (x \overset{\pi}{\mapsto} a * y \overset{\sigma}{\mapsto} b) \vee (x = y \wedge a = b \wedge x \overset{\pi \oplus \sigma}{\mapsto} a)$$

In other words, either $x$ and $y$ are not aliased, or they *are* aliased and the permissions combine (the additive operation $\oplus$ on rational shares is simply normal addition when the sum is $\leq 1$ and undefined otherwise). This disjunction undermines the notational economies that have led to separation logic's great successes in scalable verification [11]; in particular, (I) fails because the left disjunct might be true, and (II) fails because the right disjunct might be. At a high level, $\circledast$ is a bit too easy to introduce, and therefore also a bit too hard to eliminate.

### 2.1 Weak vs. strong separation and the distribution principle

One of the challenges of the weak separating conjunction $\circledast$ is that it interacts poorly with inductively defined predicates. Consider porting the usual separation logic definition of a possibly-cyclic linked list segment from $x$ to $y$ from a sequential setting to a concurrent one by a simple substitution of $\circledast$ for $*$:

$$\mathsf{ls}\, x\, y \;=_{\text{def}}\; (x = y \wedge \mathsf{emp}) \vee (\exists z.\; x \mapsto z \circledast \mathsf{ls}\, z\, y)\ .$$

Now consider a simple recursive procedure `foo(x,y)` that traverses a linked list segment from `x` to `y`:

```
foo(x,y) { if x=y then return; else foo([x],y); }
```

It is easy to see that `foo` leaves the list segment unchanged, and therefore satisfies the following Hoare triple:

$$\{(\mathsf{ls}\, x\, y)^{0.5}\}\ \texttt{foo(x,y);}\ \{(\mathsf{ls}\, x\, y)^{0.5}\}\ .$$

The intuitive proof of this fact would run approximately as follows:

$\{(\mathsf{ls}\, x\, y)^{0.5}\}$ `foo(x,y) {`

`  if x=y then return;` $\{(\mathsf{ls}\, x\, y)^{0.5}\}$

`  else` $\{x \neq y \wedge (x \mapsto z \circledast \mathsf{ls}\, z\, y)^{0.5}\}$

$\{x \overset{0.5}{\mapsto} z \circledast (\mathsf{ls}\, z\, y)^{0.5}\}$

`  foo([x],y);` $\{x \overset{0.5}{\mapsto} z \circledast (\mathsf{ls}\, z\, y)^{0.5}\}$

$\{(x \mapsto z \circledast \mathsf{ls}\, z\, y)^{0.5}\}$

$\{(\mathsf{ls}\, x\, y)^{0.5}\}$

`}` $\{(\mathsf{ls}\, x\, y)^{0.5}\}$

However, because of the use of $\circledast$, the highlighted inference step is not sound:

$$x \overset{0.5}{\mapsto} z \circledast (\mathsf{ls}\, z\, y)^{0.5} \not\models (x \mapsto z \circledast \mathsf{ls}\, z\, y)^{0.5}\ . \tag{2}$$

To see this, consider a heap with the following structure, viewed in two ways:

$$x \overset{0.5}{\mapsto} z \circledast z \overset{0.5}{\mapsto} x \circledast x \overset{0.5}{\mapsto} z \quad = \quad x \mapsto z \circledast z \overset{0.5}{\mapsto} x$$

This heap satisfies the LHS of the entailment in (2), as it is the $\circledast$-composition of a 0.5-share of $x \mapsto z$ and a 0.5-share of $\mathsf{ls}\, z\, z$, a cyclic list segment from $z$ back to itself (note that here $z = y$). However, it does not satisfy the RHS, since it is not a 0.5-share of the $\circledast$-composition of $x \mapsto z$ with $\mathsf{ls}\, z\, z$, which would require the pointer to be disjoint from the list segment.

The underlying reason for the failure of this example is that, in going from $(x \mapsto z \circledast \mathsf{ls}\, z\, z)^{0.5}$ to $x \overset{0.5}{\mapsto} z \circledast (\mathsf{ls}\, z\, z)^{0.5}$, we have lost the information that the pointer and the list segment are actually disjoint. This is reflected in the general failure of the distribution principle $A^\pi \circledast B^\pi \models (A \circledast B)^\pi$, of which the above is just one instance. Accordingly, our proposal is that the "strong" separating conjunction $*$ from standard separation logic, which forces disjointness of the heaps satisfying its conjuncts, should *also* be retained in the logic alongside $\circledast$, on the grounds that (II) *is* true for the stronger connective:

$$(A * B)^\pi \equiv A^\pi * B^\pi \ . \tag{3}$$

If we then define our list segments using $*$ in the traditional way, namely

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

then we can observe that this second definition of $\mathsf{ls}$ is identical to the first on permission-free formulas, since $\circledast$ and $*$ coincide in that case. However, when we replay the verification proof above with the new definition of $\mathsf{ls}$, every $\circledast$ in the proof above becomes a $*$, and the proof then becomes sound. Nevertheless, we can still use $\circledast$ to describe permission-decomposition of list segments at a higher level; e.g., $\mathsf{ls}\, x\, y$ can still be decomposed as $(\mathsf{ls}\, x\, y)^{0.5} \circledast (\mathsf{ls}\, x\, y)^{0.5}$.

### 2.2 Nominal labelling and the combination principle

Unfortunately, even when we use the strong separating conjunction $*$ to define list segments $\mathsf{ls}$, a further difficulty still remains. Consider a simple concurrent program that runs two copies of `foo` in parallel on the same list segment:

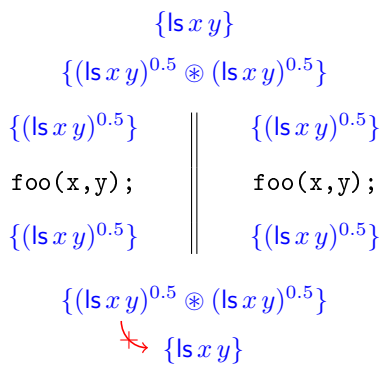
```
foo(x,y); || foo(x,y);
```

Since `foo` only reads from its input list segment, and satisfies the specification $\{(\mathsf{ls}\, x\, y)^{0.5}\}$`foo(x,y);`$\{(\mathsf{ls}\, x\, y)^{0.5}\}$, this program satisfies the specification

$$\{\mathsf{ls}\, x\, y\}\texttt{foo(x,y); || foo(x,y);}\{\mathsf{ls}\, x\, y\} \ .$$

Now consider constructing a proof of this specification in CSL. First we view the list segment $\mathsf{ls}\, x\, y$ as the $\circledast$-composition of two read-only copies, with permission 0.5 each; then we use CSL's concurrency rule (see Section 1) to compose the

specifications of the two threads; last we recombine the two read-only copies to obtain the original list segment. The proof diagram is as follows:




However, again, the highlighted inference step in this proof is not correct:

$$(\mathsf{ls}\, x\, y)^{0.5} \circledast (\mathsf{ls}\, x\, y)^{0.5} \not\models \mathsf{ls}\, x\, y\ . \tag{4}$$

A countermodel is a heap with the following structure, again viewed in two ways:

$$(x \overset{0.5}{\mapsto} y \circledast y \overset{0.5}{\mapsto} y) \circledast x \overset{0.5}{\mapsto} y \quad = \quad x \mapsto y \circledast y \overset{0.5}{\mapsto} y$$

According to the first view of such a heap, it satisfies the LHS of (4), as it is the $\circledast$-composition of two 0.5-shares of $\mathsf{ls}\, x\, y$ (one of two cells, and one of a single cell). However, it does not satisfy $\mathsf{ls}\, x\, y$, since that would require every cell in the heap to be owned with permission 1.

Like in our previous example, the reason for the failure of this example is that we have lost information. In going from $\mathsf{ls}\, x\, y$ to $(\mathsf{ls}\, x\, y)^{0.5} \circledast (\mathsf{ls}\, x\, y)^{0.5}$, we have forgotten that the two formulas $(\mathsf{ls}\, x\, y)^{0.5}$ are in fact *copies of the same region*. For formulas $A$ that are *precise* in that they uniquely describe part of any given heap [12], e.g. formulas $x \mapsto a$, this loss of information does not happen and we do have $A^{0.5} \circledast A^{0.5} \models A$; but for non-precise formulas such as $\mathsf{ls}\, x\, y$, this principle fails.

However, we regard this primarily as a technical shortcoming of the formalism, rather than a failure of our intuition. It *ought* to be true that we can take any region of memory, split it into two read-only copies, and then later merge the two copies to re-obtain the original region. Were we conducting the above proof on pen and paper, we would very likely explain the difficulty away by adopting some kind of labelling convention, allowing us to remember that two formulas have been obtained from the same memory region by dividing permissions.

In fact, that is almost exactly our proposed remedy to the situation. We introduce *nominals*, or *labels*, from hybrid logic, where a nominal $\alpha$ is interpreted as denoting a unique heap. Any formula of the form $\alpha \wedge A$ is then precise (in the above sense), and so obeys the combination principle

$$(\alpha \wedge A)^{\pi} \circledast (\alpha \wedge A)^{\sigma} \models (\alpha \wedge A)^{\sigma \oplus \pi}\ , \tag{5}$$

where $\oplus$ is addition on permissions. Thus we can repair the faulty CSL proof above by replacing every instance of the formula $\mathsf{ls}\,x\,y$ by the "labelled" formula $\alpha \wedge \mathsf{ls}\,x\,y$ (and adding an initial step in which we introduce the fresh label $\alpha$).

### 2.3 The jump modality

However, this is not quite the end of the story. Readers may have noticed that replacing $\mathsf{ls}\,x\,y$ by the "labelled" version $\alpha \wedge \mathsf{ls}\,x\,y$ also entails establishing a slightly stronger specification for the function `foo`, namely:

$$\{(\alpha \wedge \mathsf{ls}\,x\,y)^{0.5}\}\ \texttt{foo(x,y);}\ \{(\alpha \wedge \mathsf{ls}\,x\,y)^{0.5}\}\ .$$

This introduces an extra difficulty in the proof (cf. Section 2.1); at the recursive call to `foo([x],y)`, the precondition now becomes $\alpha^{0.5} \wedge (x \overset{0.5}{\mapsto} z * (\mathsf{ls}\,z\,y)^{0.5}))$, which means that we cannot apply separation logic's *frame rule* [32] to the pointer formula without first weakening away the label-share $\alpha^{0.5}$.

For this reason, we shall also employ hybrid logic's "jump" modality $@_{\_}$, where the formula $@_\alpha A$ means that $A$ is true of the heap denoted by the label $\alpha$. In the above, we can introduce labels $\beta$ and $\gamma$ for the list components $x \mapsto z$ and $\mathsf{ls}\,z\,y$ respectively, whereby we can represent the decomposition of the list by the assertion $@_\alpha(\beta * \gamma)$. Since this is a *pure* assertion that does not depend on the heap, it can be safely maintained when applying the frame rule, and used after the function call to restore the label $\alpha$, using the easily verifiable fact that

$$@_\alpha(\beta * \gamma) \wedge (\beta * \gamma) \models \alpha.$$

Similar reasoning over labelled decompositions of data structures is seemingly necessary whenever treating recursion; we return to it in more detail in Section 5.

## 3 Separation logic with labels and permissions ($\mathsf{SL_{LP}}$)

Following the motivation given in the previous section, here we give the syntax and semantics of a separation logic, $\mathsf{SL_{LP}}$, with permissions over arbitrary formulas, making use of both strong *and* weak separating conjunctions, and nominal labels (from hybrid logic [3, 10]). First, we define a suitable notion of permissions and associated operations.

**Definition 3.1.** *A* permissions algebra *is a tuple* $\langle \mathsf{Perm}, \oplus, \otimes, 1 \rangle$*, where* $\mathsf{Perm}$ *is a set (of "permissions"),* $1 \in \mathsf{Perm}$ *is called the* write permission*, and* $\oplus$ *and* $\otimes$ *are respectively partial and total binary functions on* $\mathsf{Perm}$*, satisfying associativity, commutativity, cancellativity and the following additional axioms:*

| | |
|---|---|
| $\pi_1 \oplus \pi_2 \neq \pi_2$ | *(non-zero)* |
| $\forall \pi.\ \pi \oplus 1$ *is undefined* | *(top)* |
| $\forall \pi.\ \exists \pi_1, \pi_2.\ \pi = \pi_1 \oplus \pi_2$ | *(divisibility)* |
| $(\pi_1 \oplus \pi_2) \otimes \pi = (\pi_1 \otimes \pi) \oplus (\pi_2 \otimes \pi)$ | *(left-dist)* |

The most common example of a permissions algebra is the Boyland fractional permission model $\langle (0,1] \cap \mathbb{Q}, \oplus, \times, 1 \rangle$, where permissions are rational numbers in $(0,1]$, $\times$ is standard multiplication, and $\oplus$ is standard addition but undefined if $p + p' > 1$. From now on, we assume a fixed but arbitrary permissions algebra.

With the permissions structure in place, we can now define the syntax of our logic. We assume disjoint, countably infinite sets $\mathsf{Var}$ of variables, $\mathsf{Pred}$ of predicate symbols (with associated arities) and $\mathsf{Label}$ of labels.

**Definition 3.2.** *We define* formulas *of* $\mathsf{SL}_{\mathsf{LP}}$ *by the grammar:*

$$\begin{array}{lll} A ::= x = y \mid \neg A \mid A \wedge A \mid A \vee A \mid A \to A & & \textit{(pure)} \\ \quad \mid \mathsf{emp} \mid x \mapsto y \mid P(\mathbf{x}) \mid A * A \mid A \circledast A \mid A \mathrel{-\!\!*} A \mid A \mathrel{-\!\!\circledast} A & & \textit{(spatial)} \\ \quad \mid A^{\pi} \mid \alpha \mid @_{\alpha} A & & \textit{(perms / labels)} \end{array}$$

*where $x, y$ range over $\mathsf{Var}$, $\pi$ ranges over $\mathsf{Perm}$, $P$ ranges over $\mathsf{Pred}$, $\alpha$ ranges over $\mathsf{Label}$ and $\mathbf{x}$ ranges over tuples of variables of length matching the arity of the predicate symbol $P$. We write $x \overset{\pi}{\mapsto} y$ for $(x \mapsto y)^{\pi}$, and $x \neq y$ for $\neg(x = y)$.*

The "magic wands" $\mathrel{-\!\!*}$ and $\mathrel{-\!\!\circledast}$ are the implications adjoint to $*$ and $\circledast$, as usual in separation logic. We include them for completeness, but we use $\mathrel{-\!\!*}$ only for fairly complex examples (see Section 5.3) and in fact do not use $\mathrel{-\!\!\circledast}$ at all.

*Semantics.* We interpret formulas in a standard model of stacks and heaps-with-permissions (cf. [4]), except that our models also incorporate a valuation of nominal labels. We assume an infinite set $\mathsf{Val}$ of *values* of which an infinite subset $\mathsf{Loc} \subset \mathsf{Val}$ are considered addressable *locations*. A *stack* is as usual a map $s : \mathsf{Var} \to \mathsf{Val}$. A *heap-with-permissions*, which we call a *p-heap* for short, is a finite partial function $h : \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val} \times \mathsf{Perm}$ from locations to value-permission pairs. We write $\mathrm{dom}\,(h)$ for the *domain* of $h$, i.e. the set of locations on which $h$ is defined. Two p-heaps $h_1$ and $h_2$ are called *disjoint* if $\mathrm{dom}\,(h_1) \cap \mathrm{dom}\,(h_2) = \emptyset$, and *compatible* if, for all $\ell \in \mathrm{dom}\,(h_1) \cap \mathrm{dom}\,(h_2)$, we have $h_1(\ell) = (v, \pi_1)$ and $h_2(v, \pi_2)$ and $\pi_1 \oplus \pi_2$ is defined. (Thus, trivially, disjoint heaps are also compatible.) We define the multiplication $\pi \cdot h$ of a p-heap $h$ by permission $\pi$ by extending $\otimes$ pointwise:

$$(\pi \cdot h)(\ell) = (v, \pi \otimes \pi') \quad \Leftrightarrow \quad h(\ell) = (v, \pi') \ .$$

We also assume that each predicate symbol $P$ of arity $k$ is given a fixed interpretation $[\![P]\!] \in (\mathsf{Val}^k \times \mathsf{PHeaps})$, where $\mathsf{PHeaps}$ is the set of all p-heaps. Here we allow an essentially free interpretation of predicate symbols, but they could also be given by a suitable inductive definition schema, as is done in many papers on separation logic (e.g. [7, 8]). Finally, a *valuation* is a function $\rho : \mathsf{Label} \to \mathsf{PHeaps}$ assigning a single p-heap $\rho(\alpha)$ to each label $\alpha$.

**Definition 3.3 (Strong and weak heap composition).** *The* strong composition $h_1 \circ h_2$ *of two disjoint p-heaps $h_1$ and $h_2$ is defined as their union:*

$$(h_1 \circ h_2)(\ell) = \begin{cases} h_1(\ell) & \textit{if } \ell \notin \mathrm{dom}\,(h_2) \\ h_2(\ell) & \textit{if } \ell \notin \mathrm{dom}\,(h_1) \end{cases}$$

$$
\begin{array}{lcl}
s,h,\rho \models x = y & \Leftrightarrow & s(x) = s(y) \\
s,h,\rho \models \neg A & \Leftrightarrow & s,h,\rho \not\models A \\
s,h,\rho \models A \wedge B & \Leftrightarrow & s,h,\rho \models A \text{ and } s,h,\rho \models B \\
s,h,\rho \models A \vee B & \Leftrightarrow & s,h,\rho \models A \text{ or } s,h,\rho \models B \\
s,h,\rho \models A \to B & \Leftrightarrow & s,h,\rho \models A \text{ implies } s,h,\rho \models B \\
s,h,\rho \models \mathsf{emp} & \Leftrightarrow & \mathrm{dom}\,(h) = \emptyset \\
s,h,\rho \models x \mapsto y & \Leftrightarrow & \mathrm{dom}\,(h) = \{s(x)\} \text{ and } h(s(x)) = (s(y),1) \\
s,h,\rho \models P(\mathbf{x}) & \Leftrightarrow & (s(\mathbf{x}),h) \in [\![P]\!] \\
s,h,\rho \models A * B & \Leftrightarrow & \exists h_1,h_2.\ h = h_1 \circ h_2 \text{ and } s,h_1,\rho \models A \text{ and } s,h_2,\rho \models B \\
s,h,\rho \models A \circledast B & \Leftrightarrow & \exists h_1,h_2.\ h = h_1 \mathbin{\overline{\circ}} h_2 \text{ and } s,h_1,\rho \models A \text{ and } s,h_2,\rho \models B \\
s,h,\rho \models A \mathrel{-\!\!*} B & \Leftrightarrow & \forall h'.\ \text{if } h \circ h' \text{ defined and } s,h',\rho \models A \text{ then } s,h \circ h',\rho \models B \\
s,h,\rho \models A \mathrel{-\!\!\circledast} B & \Leftrightarrow & \forall h'.\ \text{if } h \mathbin{\overline{\circ}} h' \text{ defined and } s,h',\rho \models A \text{ then } s,h \mathbin{\overline{\circ}} h',\rho \models B \\
s,h,\rho \models A^{\pi} & \Leftrightarrow & \exists h'.\ h = \pi \cdot h' \text{ and } s,h',\rho \models A \\
s,h,\rho \models \alpha & \Leftrightarrow & h = \rho(\alpha) \\
s,h,\rho \models @_{\alpha} A & \Leftrightarrow & s,\rho(\alpha),\rho \models A
\end{array}
$$

**Fig. 1.** Definition of the satisfaction relation $s,h,\rho \models A$ for $\mathsf{SL}_{\mathsf{LP}}$.

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

*The* weak composition *$h_1 \mathbin{\overline{\circ}} h_2$ of two compatible p-heaps $h_1$ and $h_2$ is defined as their union, adding permissions at overlapping locations:*

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

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

**Definition 3.4.** *The satisfaction relation $s,h,\rho \models A$, where $s$ is a stack, $h$ a p-heap, $\rho$ a valuation and $A$ a formula, is defined by structural induction on $A$ in Figure 1. We write the* entailment *$A \models B$, where $A$ and $B$ are formulas, to mean that if $s,h,\rho \models A$ then $s,h,\rho \models B$. We write the* equivalence *$A \equiv B$ to mean that $A \models B$ and $B \models A$.*

## 4 Logical principles of $\mathsf{SL}_{\mathsf{LP}}$

In this section, we establish the main logical entailments and equivalences of $\mathsf{SL}_{\mathsf{LP}}$ that capture the various interactions between the separating conjunctions $\circledast$ and $*$, permissions and labels. As well as being of interest in their own right, many of these principles will be essential in treating the practical verification examples in Section 5. In particular, the permission distribution principle for $*$ (cf. (3), Section 2) is given in Lemma 4.3, and the permission combination principle for labelled formulas (cf. (5), Section 2) is given in Lemma 4.4.

**Proposition 4.1.** *The following equivalences all hold in* $\mathsf{SL}_{\mathsf{LP}}$*:*

$$\begin{array}{cc} A \circledast B \equiv B \circledast A & A * B \equiv B * A \\ A \circledast (B \circledast C) \equiv (A \circledast B) \circledast C & A * (B * C) \equiv (A * B) * C \\ A \circledast \mathsf{emp} \equiv A & A * \mathsf{emp} \equiv A \end{array}$$

*Additionally, the following residuation laws hold:*

$$A \models B \mathrel{-\!\!\circledast} C \;\Leftrightarrow\; A \circledast B \models C \quad \textit{and} \quad A \models B \mathrel{-\!\!*} C \;\Leftrightarrow\; A * B \models C \;.$$

*In addition, we can always weaken* $*$ *to* $\circledast$*:* $A * B \models A \circledast B$*.*

Next, we establish an additional connection between the two separating conjunctions $\circledast$ and $*$.

**Lemma 4.2 ($\circledast/*$ distribution).** *For all formulas A, B, C and D,*

$$(A \circledast B) * (C \circledast D) \models (A * C) \circledast (B * D) \;. \qquad (\circledast/*)$$

*Proof.* First we show a corresponding model-theoretic property: for any p-heaps $h_1, h_2, h_3$ and $h_4$ such that $(h_1 \mathbin{\overline{\circ}} h_2) \circ (h_3 \mathbin{\overline{\circ}} h_4)$ is defined,

$$(h_1 \mathbin{\overline{\circ}} h_2) \circ (h_3 \mathbin{\overline{\circ}} h_4) = (h_1 \circ h_3) \mathbin{\overline{\circ}} (h_2 \circ h_4) \qquad (6)$$

Since $(h_1 \mathbin{\overline{\circ}} h_2) \circ (h_3 \mathbin{\overline{\circ}} h_4)$ is defined by assumption, we have that $h_1 \mathbin{\overline{\circ}} h_2$ and $h_3 \mathbin{\overline{\circ}} h_4$ are disjoint and that $h_1$ and $h_2$, as well as $h_3$ and $h_4$ are compatible. In particular, $h_1$ and $h_3$ are disjoint, so $h_1 \circ h_3$ is defined; the same reasoning applies to $h_2$ and $h_4$. Moreover, since $h_1$ and $h_2$ are compatible, $h_1 \circ h_3$ and $h_2 \circ h_4$ must be compatible and so $(h_1 \circ h_3) \mathbin{\overline{\circ}} (h_2 \circ h_4)$ is defined.
Now, writing $h$ for $(h_1 \mathbin{\overline{\circ}} h_2) \circ (h_3 \mathbin{\overline{\circ}} h_4)$, and letting $\ell \in \mathrm{dom}\,(h)$, we have

$$h(\ell) = \begin{cases} h_1(\ell) & \text{if } \ell \notin \mathrm{dom}\,(h_3), \ell \notin \mathrm{dom}\,(h_4) \text{ and } \ell \notin \mathrm{dom}\,(h_2) \\ h_2(\ell) & \text{if } \ell \notin \mathrm{dom}\,(h_3), \ell \notin \mathrm{dom}\,(h_4) \text{ and } \ell \notin \mathrm{dom}\,(h_1) \\ (v, \pi_1 \oplus \pi_2) & \text{if } \ell \notin \mathrm{dom}\,(h_3), \ell \notin \mathrm{dom}\,(h_4) \text{ and } h_1(\ell) = (v, \pi_1) \\ & \text{and } h_2(\ell) = (v, \pi_2) \\ h_3(\ell) & \text{if } \ell \notin \mathrm{dom}\,(h_1), \ell \notin \mathrm{dom}\,(h_2) \text{ and } \ell \notin \mathrm{dom}\,(h_4) \\ h_4(\ell) & \text{if } \ell \notin \mathrm{dom}\,(h_1), \ell \notin \mathrm{dom}\,(h_2) \text{ and } \ell \notin \mathrm{dom}\,(h_3) \\ (u, \pi_3 \oplus \pi_4) & \text{if } \ell \notin \mathrm{dom}\,(h_1), \ell \notin \mathrm{dom}\,(h_2) \text{ and } h_3(\ell) = (u, \pi_3) \\ & \text{and } h_4(\ell) = (u, \pi_4) \end{cases}$$

We can merge the first and fourth cases by noting that $h(\ell) = (h_1 \circ h_3)(\ell)$ if $\ell \notin \mathrm{dom}\,(h_2 \circ h_4)$, and similarly for the second and fifth cases. We can also rewrite the last two cases by observing that $\ell \notin \mathrm{dom}\,(h_3)$ implies $h_1(\ell) = (h_1 \circ h_3)(\ell)$, and so on, resulting in

$$h(\ell) = \begin{cases} (h_1 \circ h_3)(\ell) & \text{if } \ell \notin \mathrm{dom}\,(h_2 \circ h_4) \\ (h_2 \circ h_4)(\ell) & \text{if } \ell \notin \mathrm{dom}\,(h_1 \circ h_3) \\ (w, \sigma_1 \oplus \sigma_2) & \text{if } (h_1 \circ h_3)(\ell) = (w, \sigma_1) \text{ and } (h_2 \circ h_4)(\ell) = (w, \sigma_2) \end{cases}$$

$$= ((h_1 \circ h_3) \mathbin{\overline{\circ}} (h_2 \circ h_4))(\ell) \;.$$

Now we show the main result. Suppose $s, h, \rho \models (A \circledast B) * (C \circledast D)$. This gives us $h = (h_1 \mathbin{\overline{\circ}} h_2) \circ (h_3 \mathbin{\overline{\circ}} h_4)$, where $s, h_1, \rho \models A$ and $s, h_2, \rho \models B$ and $s, h_3, \rho \models C$ and $s, h_4, \rho \models D$. By equation (6), we have $h = (h_1 \circ h_3) \mathbin{\overline{\circ}} (h_2 \circ h_4)$, which gives us exactly that $s, h, \rho \models (A * C) \circledast (B * D)$, as required. □

Next, we establish principles for distributing permissions over various connectives, in particular over the strong $*$, stated earlier as (3) in Section 2.

**Lemma 4.3 (Permission distribution).** *The following equivalences hold for all formulas $A$ and $B$, and permissions $\pi$ and $\sigma$:*

$$\left(A^{\sigma}\right)^{\pi} \equiv A^{\sigma \otimes \pi} \tag{$\otimes$}$$
$$(A \vee B)^{\pi} \equiv A^{\pi} \vee B^{\pi} \tag{$\vee^{\pi}$}$$
$$(A \wedge B)^{\pi} \equiv A^{\pi} \wedge B^{\pi} \tag{$\wedge^{\pi}$}$$
$$(A * B)^{\pi} \equiv A^{\pi} * B^{\pi} \tag{$*^{\pi}$}$$

*Proof.* We just show the most interesting case, $(*^{\pi})$. First of all, we establish a corresponding model-theoretic property: for any permission $\pi$ and disjoint p-heaps $h_1$ and $h_2$, meaning $h_1 \circ h_2$ is defined,

$$\pi \cdot (h_1 \circ h_2) = (\pi \cdot h_1) \circ (\pi \cdot h_2) \ . \tag{7}$$

To see this, we first observe that for any $\ell \in \mathrm{dom}\,(h_1 \circ h_2)$, we have that either $\ell \in \mathrm{dom}\,(h_1)$ or $\ell \in \mathrm{dom}\,(h_2)$. We just show the case $\ell \in \mathrm{dom}\,(h_1)$, since the other is symmetric. Writing $h_1(\ell) = (v_1, \pi_1)$, and using the fact that $\ell \notin \mathrm{dom}\,(h_2)$,

$$\pi \cdot (h_1 \circ h_2)(\ell) = (v_1, \pi \otimes \pi_1) = (\pi \cdot h_1)(\ell) = ((\pi \cdot h_1) \circ (\pi \cdot h_2))(\ell) \ .$$

Now for the main result, let $s$, $h$ and $\rho$ be given. We have

$$\begin{array}{lll}
 & s, h, \rho \models (A * B)^{\pi} & \\
\Leftrightarrow & h = \pi \cdot h' \text{ and } s, h', \rho \models A * B & \\
\Leftrightarrow & h = \pi \cdot h' \text{ and } h' = h_1 \circ h_2 \text{ and } s, h_1, \rho \models A \text{ and } s, h_2, \rho \models B & \\
\Leftrightarrow & h = \pi \cdot (h_1 \circ h_2) \text{ and } s, h_1, \rho \models A \text{ and } s, h_2, \rho \models B & \\
\Leftrightarrow & h = (\pi \cdot h_1) \circ (\pi \cdot h_2) \text{ and } s, h_1, \rho \models A \text{ and } s, h_2, \rho \models B & \text{by (7)} \\
\Leftrightarrow & h = h_1' \circ h_2' \text{ and } s, h_1', \rho \models A^{\pi} \text{ and } s, h_2', \rho \models B^{\pi} & \\
\Leftrightarrow & s, h, \rho \models A^{\pi} * B^{\pi} \ . &
\end{array}$$

□

We now establish the main principles for dividing and combining permissions formulas using $\circledast$. As foreshadowed in Section 2, the combination principle holds only for formulas that are conjoined with a nominal label (cf. equation (5)).

**Lemma 4.4 (Permission division and combination).** *For all formulas $A$, nominals $\alpha$, and permissions $\pi_1, \pi_2$ such that $\pi_1 \oplus \pi_2$ is defined:*

$$A^{\pi_1 \oplus \pi_2} \models A^{\pi_1} \circledast A^{\pi_2} \tag{Split $\circledast$}$$
$$(\alpha \wedge A)^{\pi_1} \circledast (\alpha \wedge A)^{\pi_2} \models (\alpha \wedge A)^{\pi_1 \oplus \pi_2} \tag{Join $\circledast$}$$

*Proof.* **Case (**Split $\circledast$**):** Suppose that $s, h, \rho \models A^{\pi_1 \oplus \pi_2}$. We have $h = (\pi_1 \oplus \pi_2) \cdot h'$, where $s, h', \rho \models A$. That is, for any $\ell \in \mathrm{dom}\,(h)$, we have $h'(\ell) = (v, \pi)$ say and, using the permissions algebra axiom (left-dist) from Definition 3.1,

$$h(\ell) = (v, (\pi_1 \oplus \pi_2) \otimes \pi) = (v, (\pi_1 \otimes \pi) \oplus (\pi_2 \otimes \pi)) \ .$$

Now we define p-heaps $h_1$ and $h_2$, both with domain exactly $\mathrm{dom}\,(h)$, by

$$h_i(\ell) = (v, \pi_i \otimes \pi) \ \Leftrightarrow \ h'(\ell) = (v, \pi) \qquad \text{for } i \in \{1, 2\} \ .$$

By construction, $h_1 = \pi_1 \cdot h'$ and $h_2 = \pi_2 \cdot h'$. Since $s, h', \rho \models A$, this gives us $s, h_1, \rho \models A^{\pi_1}$ and $s, h_2, \rho \models A^{\pi_2}$. Furthermore, also by construction, $h_1$ and $h_2$ are compatible, with $h = h_1 \mathbin{\overline{\circ}} h_2$. Thus $s, h, \rho \models A^{\pi_1} \circledast A^{\pi_2}$, as required.

**Case (**Join $\circledast$**):** First of all, we show that for any p-heap $h$,

$$(\pi_1 \cdot h) \mathbin{\overline{\circ}} (\pi_2 \cdot h) = (\pi_1 \oplus \pi_2) \cdot h \ . \tag{8}$$

To see this, we observe that for any $\ell \in \mathrm{dom}\,(h)$, writing $h(\ell) = (v, \pi)$ say,

$$\begin{aligned}
&((\pi_1 \oplus \pi_2) \cdot h)(\ell) \\
&= (v, (\pi_1 \oplus \pi_2) \otimes \pi) \\
&= (v, (\pi_1 \otimes \pi) \oplus (\pi_2 \otimes \pi)) \qquad \text{by (left-dist)} \\
&= (h_1 \oplus h_2)(\ell) \text{ where } h_1(\ell) = (v, \pi_1 \otimes \pi) \text{ and } h_2 = (v, \pi_2 \otimes \pi) \\
&= ((\pi_1 \cdot h) \mathbin{\overline{\circ}} (\pi_2 \cdot h))(\ell) \ .
\end{aligned}$$

Now, for the main result, suppose $s, h, \rho \models (\alpha \wedge A)^{\pi_1} \circledast (\alpha \wedge A)^{\pi_2}$. We have $h = h_1 \mathbin{\overline{\circ}} h_2$ where $s, h_1, \rho \models (\alpha \wedge A)^{\pi_1}$ and $s, h_2, \rho \models (\alpha \wedge A)^{\pi_2}$. That is, $h = (\pi_1 \cdot h_1') \mathbin{\overline{\circ}} (\pi_2 \cdot h_2')$, where $s, h_1', \rho \models \alpha \wedge A$ and $s, h_2', \rho \models \alpha \wedge A$. Thus $h_1' = h_2' = \rho(\alpha)$ and so, by (8), we have $h = (\pi_1 \oplus \pi_2) \cdot h_1'$, where $s, h_1', \rho \models \alpha \wedge A$. This gives us $s, h, \rho \models (\alpha \wedge A)^{\pi_1 \oplus \pi_2}$, as required. □

Lastly, we state some useful principles for labels and the "jump" modality.

**Lemma 4.5 (Labelling and jump).** *For all formulas $A$ and labels $\alpha$,*

$$@_\alpha A \wedge \alpha^\pi \models A^\pi \qquad (@\,\text{Elim})$$

$$(\alpha \wedge A)^\pi \models @_\alpha A \qquad (@\,\text{Intro})$$

$$@_\alpha({\beta_1}^\pi * {\beta_2}^\sigma) \wedge ({\beta_1}^\pi \circledast {\beta_2}^\sigma) \models \alpha \wedge ({\beta_1}^\pi * {\beta_2}^\sigma) \qquad (@/*/\circledast)$$

*Proof.* We just show the case $(@/*/\circledast)$, the others being easy. Suppose $s, h, \rho \models @_\alpha({\beta_1}^\pi * {\beta_2}^\sigma) \wedge ({\beta_1}^\pi \circledast {\beta_2}^\sigma)$, meaning that $s, \rho(\alpha), \rho \models {\beta_1}^\pi * {\beta_2}^\sigma$ and $s, h, \rho \models {\beta_1}^\pi \circledast {\beta_2}^\sigma$. Then we have $\rho(\alpha) = (\pi \cdot \rho(\beta_1)) \circ (\sigma \cdot \rho(\beta_2))$, while $h = (\pi \cdot \rho(\beta_1)) \mathbin{\overline{\circ}} (\sigma \cdot \rho(\beta_2))$. Since $\circ$ is defined only when its arguments are disjoint p-heaps, we obtain that $h = \rho(\alpha) = (\pi \cdot \rho(\beta_1)) \circ (\sigma \cdot \rho(\beta_2))$. Thus $s, h, \rho \models \alpha \wedge ({\beta_1}^\pi * {\beta_2}^\sigma)$. □

$$\frac{\{A_1\}\,C_1\,\{B_1\} \quad \{A_2\}\,C_2\,\{B_2\}}{\{A_1 \circledast A_2\}\,C_1 \,||\, C_2\,\{B_1 \circledast B_2\}}\ (\boxplus)\ \text{(Par)} \qquad \frac{\{\alpha \wedge A\}\,C\,\{B\}}{\{A\}\,C\,\{B\}}\ (\boxtimes)\ \text{(Label)}$$

$$\frac{\{A\}\,C\,\{B\}}{\{A * F\}\,C\,\{B * F\}}\ (\dagger, \ddagger)\ \text{(Frame } *) \qquad \frac{\{A\}\,C\,\{B\}}{\{A \circledast F\}\,C\,\{B \circledast F\}}\ (\dagger)\ \text{(Frame } \circledast)$$

$(\boxplus)\ \mathrm{ModVars}(C_2) \cap \mathrm{FreeVars}(A_1, B_1) = \mathrm{ModVars}(C_1) \cap \mathrm{FreeVars}(A_2, B_2) = \emptyset$

$(\boxtimes)\ \alpha$ fresh $\quad (\dagger)\ \mathrm{ModVars}(C) \cap \mathrm{FreeVars}(F) = \emptyset \quad (\ddagger)$ see §5.3

**Fig. 2.** The key CSL proof rules used in our examples; not shown are standard rules for consequence, conditionals, load/store, etc. The fresh-labelling rule (Label) and combination of both weak (Frame $\circledast$) and strong (Frame $*$) frame rules are novel to our approach. We require weak conjunction $\circledast$ for the parallel rule (Par).

## 5 Concurrent program verification examples

In this section, we demonstrate how $\mathsf{SL}_{\mathsf{LP}}$ can be used in conjunction with the usual principles of CSL to construct verification proofs of concurrent programs, taking three examples of increasing complexity.

Our examples all operate on *binary trees* in memory, defined as usual in separation logic (again note the use of $*$ rather than $\circledast$):

$$\mathsf{tree}(x) \ =_{\mathrm{def}}\ (x = \mathit{null} \wedge \mathsf{emp}) \vee (\exists d, l, r.\ x \mapsto (d, l, r) * \mathsf{tree}(l) * \mathsf{tree}(r))\ .$$

Our proofs employ (a subset of) the standard rules of CSL—with the most important being the concurrency rule from the Introduction, the separation logic *frame rules* for both $*$ and $\circledast$, and a new rule enabling us to introduce fresh labels into the precondition of a triple (similar to the way Hoare logic usually handles existential quantifiers). These key rules are shown in Figure 2. We simplify our Hoare triple to remove elements to handle function call/return and furthermore omit the presentation of the standard collection of rules for consequence, load, store, if-then-else, assignment, etc.; readers interested in such aspects can consult [1]. Both of our frame rules have the usual side condition on modified program variables. The strong frame rule (Frame $*$) has an additional side condition that will be discussed in §5.3; until then it is trivially satisfied.

### 5.1 Parallel read

Consider the following program:

```
check(x) {
   if (x == null) { return; }
   read(x);  ||  read(x);
}
```

This is intended to be a straightforward example where we take a tree rooted at `x` and, if `x` is non-null, split into parallel threads that run the program `read` on `x`, and whose specification is $\{\alpha^\pi \wedge \mathsf{tree}(x)^\sigma\}$ `read(x)` $\{\alpha^\pi \wedge \mathsf{tree}(x)^\sigma\}$. We prove that `check` satisfies the specification $\{\mathsf{tree}(x)^\pi\}$ `check(x)` $\{\mathsf{tree}(x)^\pi\}$; the verification proof is in Figure 3. The proof makes use of the basic operations of our theory: labelling, splitting and joining. The example follows precisely these steps, starting by labelling the formula $\mathsf{tree}(x)^\pi \wedge x \neq \mathit{null}$ with $\alpha$. The concurrency rule (Par) allows us to put formulas back together after the parallel call, and the two copies $(\alpha \wedge \mathsf{tree}(x)^\pi)^{0.5}$ that were obtained are glued back together to yield $\mathsf{tree}(x)^\pi$, since they have the same label.

$\{\mathsf{tree}(x)^\pi\}$
`check(x) {`
$\{(\mathsf{tree}(x)^\pi \wedge x = \mathit{null}) \vee (\mathsf{tree}(x)^\pi \wedge x \neq \mathit{null})\}$
`if (x == null) {` $\{x = \mathit{null} \wedge \mathsf{tree}(x)^\pi\}$
`return;`
$\{\mathsf{tree}(x)^\pi\}$
`}`
$\{\alpha \wedge \mathsf{tree}(x)^\pi \wedge x \neq \mathit{null}\}$ by (Label)
$\{(\alpha \wedge \mathsf{tree}(x)^\pi)^{0.5} \circledast (\alpha \wedge \mathsf{tree}(x)^\pi)^{0.5}\}$ by (Split $\circledast$)

| | | |
|---|---|---|
| $\{(\alpha \wedge \mathsf{tree}(x)^\pi)^{0.5}\}$ | $\Vert$ | |
| $\{\alpha^{0.5} \wedge \mathsf{tree}(x)^{\pi \otimes 0.5}\}$ | $\Vert$ | by $(\wedge^\pi)$,$(\otimes)$ |
| `read(x);` | $\Vert$ ... | |
| $\{\alpha^{0.5} \wedge \mathsf{tree}(x)^{\pi \otimes 0.5}\}$ | $\Vert$ | |
| $\{(\alpha \wedge \mathsf{tree}(x)^\pi)^{0.5}\}$ | $\Vert$ | by $(\wedge^\pi)$,$(\otimes)$ |

$\{(\alpha \wedge \mathsf{tree}(x)^\pi)^{0.5} \circledast (\alpha \wedge \mathsf{tree}(x)^\pi)^{0.5}\}$ by (Par)
$\{\alpha \wedge \mathsf{tree}(x)^\pi\}$ by (Join $\circledast$)
`}`
$\{\mathsf{tree}(x)^\pi\}$

**Fig. 3.** Verification proof of program `check` in Example 5.1.

### 5.2 Parallel tree processing (Le and Hobor [24]):

Consider the following program, which was also employed as an example in [24]:

```
proc(x) {
   if (x == null) { return; }
   print(x->d);  ||  print(x->d);
   proc(x->l);   ||  proc(x->l);
   proc(x->r);   ||  proc(x->r);
}
```

This code takes a tree rooted at `x` and, if `x` is non-null, splits into parallel threads that call `proc` recursively on its left and right branches. We prove, in Fig-

ure 4, that `proc` satisfies the specification $\{\alpha \wedge \mathsf{tree}(x)^\pi\}$ `proc(x)` $\{\alpha \wedge \mathsf{tree}(x)^\pi\}$. First we unroll the definition of $\mathsf{tree}(x)$ and distribute the permission over Boolean connectives and $*$. If the tree is empty the process stops. Otherwise, we label each component with a new label and introduce the "jump" statement $@_\alpha(\beta_1 * \beta_2 * \beta_3)$, recording the decomposition of the tree into its three components. Since such statements are *pure*, i.e. independent of the heap, we can "carry" this formula along our computation without interfering with the frame rule(s). Now that every subregion is labelled, we split the formula into two copies, each with half share, but after distributing 0.5 over $*$ and $\wedge$ we end up with half shares in the labels as well. We relabel each subregion with new "whole" labels, and again introduce pure @-formulas that record the relation between the old and the new labels. At this moment we enter the parallel threads and recursively apply `proc` to the left and right subtrees of `x`. Assuming the specification of `proc` for subtrees of `x`, we then retrieve the original label $\alpha$ from the trail of crumbs left by the @-formulas. We can then recombine the $\alpha$-labelled threads using (Join $\circledast$) to arrive at the desired postcondition.

### 5.3 Cross-thread data transfer

Our previous examples involve only "isolated tank" concurrency: a program has some resources and splits them into parallel threads that do not communicate with each other before —remembering Humpty Dumpty!— ultimately re-merging. For our last example, we will show that our technique is expressive enough to handle more sophisticated kinds of sharing, in particular inter-thread coarse-grained communication. We will show that we can not only share read-only data, but in fact prove that one thread has acquired the full ownership of a structure, even when the associated root pointers are not easily exposed.

To do so, we add some communication primitives to our language, together with their associated Hoare rules. Coarse-grained concurrency such as locks, channels, and barriers have been well-investigated in various flavours of concurrent separation logic [26, 31, 19]. We will use a channel for our example in this section but with simplified rules: the Hoare rule for a channel $c$ to send message number $i$ whose message invariant is $R_i^c$ is $\{R_i^c(x)\}$ `send(`$c, x$`)` $\{$`emp`$\}$, while the corresponding rule to receive is $\{$`emp`$\}$ `receive(`$c$`)` $\{\lambda ret.\ R_i^c(ret)\}$. We ignore details such as identifying which party is allowed to send / receive at a given time [14] or the resource ownership of the channel itself [18].

These rules interact poorly with the strong frame rule from Figure 2:

$$\frac{\{A\}\,C\,\{B\}}{\{A * F\}\,C\,\{B * F\}}\ (\dagger, \ddagger)\ \text{(Frame } *\text{)} \qquad \begin{array}{l}(\dagger)\ \mathrm{ModVars}(C) \cap \mathrm{FreeVars}(F) = \emptyset \\ (\ddagger)\ C \text{ does not receive resources}\end{array}$$

The revealed side condition ($\ddagger$) means that $C$ does not contain any subcommands that "transfer in" resources, such as `unlock`, `receive`, etc.; this side condition is a bit stronger than necessary but has a simple definition and can be checked syntactically. Without ($\ddagger$), we can reach a contradiction. Assume that the current

$\{\alpha \wedge \mathsf{tree}(x)^{\pi}\}$
`proc(x) {`
$\{(\alpha \wedge (x = null \wedge \mathsf{emp})^{\pi}) \vee (\alpha \wedge (x \mapsto (d,l,r) * \mathsf{tree}(l) * \mathsf{tree}(r))^{\pi})\}$ by $(\wedge^{\pi})$,$(\vee^{\pi})$
$\{(\alpha \wedge x = null \wedge \mathsf{emp}) \vee (\alpha \wedge (x \mapsto (d,l,r) * \mathsf{tree}(l) * \mathsf{tree}(r))^{\pi})\}$
`if (x == null) {` $\{\alpha \wedge x = null \wedge \mathsf{emp}\}$
`return;`
$\{\alpha \wedge (x = null \wedge \mathsf{emp})^{\pi}\}$
$\{\alpha \wedge \mathsf{tree}(x)^{\pi}\}$
`}`
$\{\alpha \wedge (x \overset{\pi}{\mapsto} (d,l,r) * \mathsf{tree}(l)^{\pi} * \mathsf{tree}(r)^{\pi})\}$ by $(*^{\pi})$
$\{((\beta_1 \wedge x \overset{\pi}{\mapsto} (d,l,r)) * (\beta_2 \wedge \mathsf{tree}(l)^{\pi}) * (\beta_3 \wedge \mathsf{tree}(r)^{\pi})) \wedge$ $\mathbin{@}_{\alpha}(\beta_1 * \beta_2 * \beta_3)\}$ by (Label), (@ Intro)
$\{(((\beta_1^{0.5} \wedge x \overset{\pi \otimes 0.5}{\mapsto} (d,l,r)) * (\beta_2^{0.5} \wedge \mathsf{tree}(l)^{\pi \otimes 0.5}) *$ $(\beta_3^{0.5} \wedge \mathsf{tree}(r)^{\pi \otimes 0.5})) \wedge (\mathbin{@}_{\alpha}(\beta_1 * \beta_2 * \beta_3))^{0.5}) \circledast$
$(((\beta_1^{0.5} \wedge x \overset{\pi \otimes 0.5}{\mapsto} (d,l,r)) * (\beta_2^{0.5} \wedge \mathsf{tree}(l)^{\pi \otimes 0.5}) *$ $(\beta_3^{0.5} \wedge \mathsf{tree}(r)^{\pi \otimes 0.5})) \wedge (\mathbin{@}_{\alpha}(\beta_1 * \beta_2 * \beta_3))^{0.5})\}$ by (Split $\circledast$), $(*^{\pi})$, $(\wedge^{\pi})$
$\{(((\gamma_1 \wedge x \overset{\pi \otimes 0.5}{\mapsto} (d,l,r) \wedge \mathbin{@}_{\gamma_1}\beta_1^{0.5}) * (\gamma_2 \wedge \mathsf{tree}(l)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_2}\beta_2^{0.5}) *$ $(\gamma_3 \wedge \mathsf{tree}(r)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_3}\beta_3^{0.5})) \wedge \mathbin{@}_{\alpha}(\beta_1 * \beta_2 * \beta_3)) \circledast$
$(((\gamma_4 \wedge x \overset{\pi \otimes 0.5}{\mapsto} (d,l,r) \wedge \mathbin{@}_{\gamma_4}\beta_1^{0.5}) * (\gamma_5 \wedge \mathsf{tree}(l)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_5}\beta_2^{0.5}) *$ $(\gamma_6 \wedge \mathsf{tree}(r)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_6}\beta_3^{0.5})) \wedge \mathbin{@}_{\alpha}(\beta_1 * \beta_2 * \beta_3))\}$ by (Label), (@ Intro)
$\{((\gamma_1 \wedge x \overset{\pi \otimes 0.5}{\mapsto} (d,l,r) \wedge \mathbin{@}_{\gamma_1}\beta_1^{0.5}) *$ $(\gamma_2 \wedge \mathsf{tree}(l)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_2}\beta_2^{0.5}) *$ $(\gamma_3 \wedge \mathsf{tree}(r)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_3}\beta_3^{0.5})) \wedge \mathbin{@}_{\alpha}(\beta_1 * \beta_2 * \beta_3)\}$
`print(x->d);`
$\{((\gamma_1 \wedge x \overset{\pi \otimes 0.5}{\mapsto} (d,l,r) \wedge \mathbin{@}_{\gamma_1}\beta_1^{0.5}) *$ $(\gamma_2 \wedge \mathsf{tree}(l)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_2}\beta_2^{0.5}) *$ $(\gamma_3 \wedge \mathsf{tree}(r)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_3}\beta_3^{0.5})) \wedge \mathbin{@}_{\alpha}(\beta_1 * \beta_2 * \beta_3)\}$
`proc(x->l);` ...
$\{((\gamma_1 \wedge x \overset{\pi \otimes 0.5}{\mapsto} (d,l,r) \wedge \mathbin{@}_{\gamma_1}\beta_1^{0.5}) *$ $(\gamma_2 \wedge \mathsf{tree}(l)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_2}\beta_2^{0.5}) *$ $(\gamma_3 \wedge \mathsf{tree}(r)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_3}\beta_3^{0.5})) \wedge \mathbin{@}_{\alpha}(\beta_1 * \beta_2 * \beta_3)\}$
`proc(x->r);`
$\{((\gamma_1 \wedge x \overset{\pi \otimes 0.5}{\mapsto} (d,l,r) \wedge \mathbin{@}_{\gamma_1}\beta_1^{0.5}) *$ $(\gamma_2 \wedge \mathsf{tree}(l)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_2}\beta_2^{0.5}) *$ $(\gamma_3 \wedge \mathsf{tree}(r)^{\pi \otimes 0.5} \wedge \mathbin{@}_{\gamma_3}\beta_3^{0.5})) \wedge \mathbin{@}_{\alpha}(\beta_1 * \beta_2 * \beta_3)\}$
$\{((\beta_1^{0.5} \wedge x \overset{\pi \otimes 0.5}{\mapsto} (d,l,r)) * (\beta_2^{0.5} \wedge \mathsf{tree}(l)^{\pi \otimes 0.5}) *$ $(\beta_3^{0.5} \wedge \mathsf{tree}(r)^{\pi \otimes 0.5})) \wedge (\mathbin{@}_{\alpha}(\beta_1 * \beta_2 * \beta_3))^{0.5}\}$
$\{(((\beta_1 \wedge x \overset{\pi}{\mapsto} (d,l,r)) * (\beta_2 \wedge \mathsf{tree}(l)^{\pi}) *$ $(\beta_3 \wedge \mathsf{tree}(r)^{\pi})) \wedge \mathbin{@}_{\alpha}(\beta_1 * \beta_2 * \beta_3))^{0.5}\}$ by $(\wedge^{\pi})$,$(*^{\pi})$
$\{(\alpha \wedge (x \overset{\pi}{\mapsto} (d,l,r) * \mathsf{tree}(l)^{\pi} * \mathsf{tree}(r)^{\pi}))^{0.5}\}$ by $(@/*/\circledast)$
$\{(\alpha \wedge (x \overset{\pi}{\mapsto} (d,l,r) * \mathsf{tree}(l)^{\pi} * \mathsf{tree}(r)^{\pi}))^{0.5} \circledast$ $(\alpha \wedge (x \overset{\pi}{\mapsto} (d,l,r) * \mathsf{tree}(l)^{\pi} * \mathsf{tree}(r)^{\pi}))^{0.5}\}$ by (Par)
$\{\alpha \wedge (x \overset{\pi}{\mapsto} (d,l,r) * \mathsf{tree}(l)^{\pi} * \mathsf{tree}(r)^{\pi})\}$ by (Join $\circledast$)
`}`
$\{\alpha \wedge \mathsf{tree}(x)^{\pi}\}$

**Fig. 4.** Verification proof of Le and Hobor's program from [24] in Example 5.2.

```
100 void transfer(int key) {   { emp }
101                           rt* = make_tree();
102                           { tree(rt) }
    { (α ∧ tree(rt))^0.5 }                  || { (α ∧ tree(rt))^0.5 }
    tree* sub = find(rt, key)   ||  ...                              ;
    send(ch, sub)               ||  tree* sub = receive(ch)          ;
    ...                         ||  modify(sub)                      ;
    receive(ch)                 ||  send(ch, ())                     ;
    { (ϵ ∧ tree(rt))^0.5 }      ||  { (ϵ ∧ tree(rt))^0.5 }
400                           { tree(rt) }
401                           delete_tree(rt);   { emp } }
```

**Fig. 5.** Verification proof of the top and bottom of `transfer` in Example 5.3.

message invariant $R_i^c$ is $x \overset{0.5}{\mapsto} a$, which has been sent by thread B. Now thread A, which had the other half of $x \overset{0.5}{\mapsto} a$, can reason as follows:

$$\frac{\{\mathsf{emp}\}\ \texttt{receive(c)}\ \{x \overset{0.5}{\mapsto} a\}}{\{\mathsf{emp} * x \overset{0.5}{\mapsto} a\}\ \texttt{receive(c)}\ \{x \overset{0.5}{\mapsto} a * x \overset{0.5}{\mapsto} a\}}\ (\text{Frame } *),\ \text{without } (\ddagger)$$

The postcondition is a contradiction as no location strongly separates from itself. However, given (‡) the strong frame rule can be proven by induction.

The consequence of (‡), from a verification point of view, is that when resources are transferred in they arrive *weakly separated*, by ⊛, since we must use the weak frame rule around the receiving command. The troublesome issue is that this newly "arriving" state can thus ⊛-overlap awkwardly with the existing state. Fortunately, judicious use of labels can sort things out.

Consider the code in Figure 5. The basic idea is simple: we create some data at the top (line 101) and then split its ownership 50-50 to two threads. The left thread finds a subtree, and passes its half of that subtree to the right via a channel. The right thread receives the root of that subtree, and thus has full ownership of that subtree along with half-ownership of the rest of the tree. Accordingly, the right thread can modify that subtree before notifying the left subtree and passing half of the modified subtree back. After merging, full ownership of the entire tree is restored and so on line 401 the program can delete it. Figure 5 only contains the proof and line numbers for the top and bottom shared portions. The left and the right thread's proofs appear in Figure 6.

By this point the top and bottom portions of the verification are straightforward. After creating the tree $\mathsf{tree}(rt)$ at line 102, we introduce the label $\alpha$, split the formula using (Split ⊛), and then pass $(\alpha \wedge \mathsf{tree}(rt))^{0.5}$ to both threads. After the parallel execution, due to the call to `modify(sub)` in the right thread, the tree has changed in memory. Accordingly, the label for the tree must also change as indicated by the $(\epsilon \wedge \mathsf{tree}(rt))^{0.5}$ in both threads after parallel processing. These are then recombined on line 400 using the re-combination principle (Join ⊛), before the tree is deallocated via standard sequential techniques.

200 $\{\ (\alpha \wedge \mathsf{tree}(\mathtt{rt}))^{0.5}\ \}$
201 `tree* sub = find(rt, key);`
202 $\{\ (\alpha^{0.5} \wedge \mathsf{tree}(\mathtt{sub}) * (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt})))^{0.5}\ \}$
203 $\{\ (\alpha^{0.5} \wedge (\beta \wedge \mathsf{tree}(\mathtt{sub})) * (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt}))))^{0.5}\ \}$
204 $\left\{\begin{array}{l} \alpha^{0.5} \wedge ((\beta \wedge \mathsf{tree}(\mathtt{sub})) * (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt}))))^{0.5} \wedge \\ (@_\alpha^{0.5}((\beta \wedge \mathsf{tree}(\mathtt{sub})) * (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt}))))^{0.5}) \end{array}\right\}$
205 $\left\{\begin{array}{l} (\beta \wedge \mathsf{tree}(\mathtt{sub}))^{0.5} * (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt})))^{0.5} \wedge \\ (@_\alpha^{0.5}((\beta \wedge \mathsf{tree}(\mathtt{sub})) * (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt}))))^{0.5}) \end{array}\right\}$
206 $\left\{\begin{array}{l} (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt})))^{0.5} \circledast (\beta \wedge \mathsf{tree}(\mathtt{sub}))^{0.5} \wedge \\ (@_\alpha^{0.5}((\beta \wedge \mathsf{tree}(\mathtt{sub})) * (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt}))))^{0.5}) \end{array}\right\}$
207 `send(ch, sub);`
208 $\{\ (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt})))^{0.5}\ \}$
209 `...`
210 $\{\ (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt})))^{0.5}\ \}$
211 `receive(ch);`
212 $\left\{\begin{array}{l} (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt})))^{0.5} \circledast ((@_\gamma^{0.5}(\delta \wedge \mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \epsilon \wedge \mathsf{tree}(\mathtt{rt}))^{0.5}) \wedge \\ \gamma \perp \delta \wedge \delta \wedge \mathsf{tree}(\mathtt{sub})^{0.5}) \end{array}\right\}$
213 $\{\ \gamma \wedge (\delta \wedge \mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \epsilon \wedge \mathsf{tree}(\mathtt{rt}))^{0.5} \circledast \delta \wedge \mathsf{tree}(\mathtt{sub})^{0.5} \wedge\ \ \gamma \perp \delta\}$
214 $\{\ (\delta \wedge \mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \epsilon \wedge \mathsf{tree}(\mathtt{rt}))^{0.5} * \delta \wedge \mathsf{tree}(\mathtt{sub})^{0.5}\ \}$
215 $\{\ (\epsilon \wedge \mathsf{tree}(\mathtt{rt}))^{0.5}\ \}$

---

300 $\{\ (\alpha \wedge \mathsf{tree}(\mathtt{rt}))^{0.5}\ \}$
301 `...`
302 $\{\ (\alpha \wedge \mathsf{tree}(\mathtt{rt}))^{0.5}\ \}$
303 `tree* sub = receive(ch);`
304 $\left\{\begin{array}{l} (\alpha \wedge \mathsf{tree}(\mathtt{rt}))^{0.5} \circledast (\beta \wedge \mathsf{tree}(\mathtt{sub}))^{0.5} \wedge \\ (@_\alpha^{0.5}((\beta \wedge \mathsf{tree}(\mathtt{sub})) * (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt}))))^{0.5}) \end{array}\right\}$
305 $\{\ ((\beta \wedge \mathsf{tree}(\mathtt{sub})) * (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt}))))^{0.5} \circledast (\beta \wedge \mathsf{tree}(\mathtt{sub}))^{0.5}\ \}$
306 $\{\ ((\beta \wedge \mathsf{tree}(\mathtt{sub}))^{0.5} \circledast (\beta \wedge \mathsf{tree}(\mathtt{sub}))^{0.5}) * (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt})))^{0.5}\ \}$
307 $\{\ \mathsf{tree}(\mathtt{sub}) * (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt})))^{0.5}\ \}$
308 `modify(sub);`
309 $\{\ \mathsf{tree}(\mathtt{sub}) * (\gamma \wedge (\mathsf{tree}(\mathtt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\mathtt{rt})))^{0.5}\ \}$
310 $\{\ (\delta \wedge \mathsf{tree}(\mathtt{sub})) * (\gamma \wedge ((\delta \wedge \mathsf{tree}(\mathtt{sub})) \mathrel{-\!\!*} (\epsilon \wedge \mathsf{tree}(\mathtt{rt}))))^{0.5} \wedge \gamma \perp \delta\ \}$
311 $\left\{\begin{array}{l} ((\delta \wedge \mathsf{tree}(\mathtt{sub}))^{0.5} \circledast (\delta \wedge \mathsf{tree}(\mathtt{sub}))^{0.5}) * (\gamma \wedge ((\delta \wedge \mathsf{tree}(\mathtt{sub})) \mathrel{-\!\!*} (\epsilon \wedge \mathsf{tree}(\mathtt{rt}))))^{0.5} \wedge \\ \gamma \perp \delta \wedge (@_\gamma^{0.5}((\delta \wedge \mathsf{tree}(\mathtt{sub})) \mathrel{-\!\!*} (\epsilon \wedge \mathsf{tree}(\mathtt{rt})))^{0.5}) \end{array}\right\}$
312 $\left\{\begin{array}{l} ((\delta \wedge \mathsf{tree}(\mathtt{sub}))^{0.5} * (\gamma \wedge ((\delta \wedge \mathsf{tree}(\mathtt{sub})) \mathrel{-\!\!*} (\epsilon \wedge \mathsf{tree}(\mathtt{rt}))))^{0.5}) \circledast \\ (\delta \wedge \mathsf{tree}(\mathtt{sub}))^{0.5} \wedge \gamma \perp \delta \wedge (@_\gamma^{0.5}((\delta \wedge \mathsf{tree}(\mathtt{sub})) \mathrel{-\!\!*} (\epsilon \wedge \mathsf{tree}(\mathtt{rt})))^{0.5}) \end{array}\right\}$
313 $\left\{\begin{array}{l} (\epsilon \wedge \mathsf{tree}(\mathtt{rt}))^{0.5} \circledast \\ (\delta \wedge \mathsf{tree}(\mathtt{sub}))^{0.5} \wedge \gamma \perp \delta \wedge (@_\gamma^{0.5}((\delta \wedge \mathsf{tree}(\mathtt{sub})) \mathrel{-\!\!*} (\epsilon \wedge \mathsf{tree}(\mathtt{rt})))^{0.5}) \end{array}\right\}$
314 `send(ch, ());`
315 $\{\ (\epsilon \wedge \mathsf{tree}(\mathtt{rt}))^{0.5}\ \}$

**Fig. 6.** Verifications of the left (top) and right (bottom) threads of `transfer`.

Let us now examine the more interesting proofs of the individual threads in Figure 6. Line 201 calls the `find` function, which searches a binary tree for a subtree rooted with key `key`. Following Cao *et al.* [13] we specify `find` as follows:

$$\{\ \mathsf{tree}(x)^{\pi}\ \}\ \texttt{find}(x)\ \{\ \lambda ret.\ \left(\mathsf{tree}(ret) * (\mathsf{tree}(ret) \mathrel{-\!\!*} \mathsf{tree}(x))\right)^{\pi}\ \}$$

Here *ret* is bound to the return value of `find`, and the postcondition can be considered to represent the returned subtree $\mathsf{tree}(ret)$ separately from the tree-with-a-hole $\mathsf{tree}(ret) \mathrel{-\!\!*} \mathsf{tree}(x)$, using a $*/\mathrel{-\!\!*}$ style to represent replacement as per Hobor and Villard [20]. This is the invariant on line 202.

Line 203 then attaches the fresh labels $\beta$ and $\gamma$ to the $*$-separated subparts, and line 204 snapshots the formula current at label $\alpha$ using the @ operator; $@^{\pi}_{\alpha} P$ should be read as "when one has a $\pi$-fraction of $\alpha$, $P$ holds"; it is definable using @ and an existential quantifier over labels. On line 205 we forget (in the left thread) the label $\alpha$ for the current heap for housekeeping purposes, and then on line 206 we weaken the strong separating conjunction $*$ to the weak one $\circledast$ before sending the root of the subtree `sub` on line 207.

In the `transfer` program, the invariant for the first channel message is

$$(\beta \wedge \mathsf{tree}(\texttt{sub}))^{0.5} \wedge \left(@^{0.5}_{\alpha}((\beta \wedge \mathsf{tree}(\texttt{sub})) * (\gamma \wedge (\mathsf{tree}(\texttt{sub}) \mathrel{-\!\!*} \mathsf{tree}(\texttt{rt}))))^{0.5}\right)$$

In other words, half of the ownership of the tree rooted at `sub` plus the (pure) @-fact about the shape of the heap labeled by $\alpha$. Comparing lines 206 and 208 we can see that this information has been shipped over the wire (the @-information has been dropped since no longer needed). The left thread then continues to process until synchronizing again with the `receive` in line 211.

Before we consider the second synchronization, however, let us instead jump to the corresponding receive in the right thread at line 303. After the receive, the invariant on line 304 has the (weakly separated) resources sent from the left thread on line 206. We then "jump" label $\alpha$ using the @-information to reach line 305. We can redistribute the $\beta$ inside the $*$ on line 306 since we already know that $\beta$ and $\gamma$ are disjoint. On line 307 we reach the payoff by combining both halves of the subtree `sub`, enabling the modification of the subtree in line 308.

On line 310 we label the two subheaps, and specialize the magic wand so that given the specific heap $\delta$ it will yield the specific heap $\epsilon$; we also record the pure fact that $\gamma$ and $\delta$ are disjoint, written $\gamma \perp \delta$. On line 311 we snapshot $\gamma$ and split the tree `sub` 50-50; then on line 312 we push half of `sub` out of the strong $*$. On line 313 we combine the subtree and the tree-with-hole to reach the final tree $\epsilon$. We then send on line 314 with the channel's second resource invariant:

$$(\delta \wedge \mathsf{tree}(\texttt{sub}))^{0.5} \wedge \gamma \perp \delta \wedge \left(@^{0.5}_{\gamma}((\delta \wedge \mathsf{tree}(\texttt{sub})) \mathrel{-\!\!*} (\epsilon \wedge \mathsf{tree}(\texttt{rt})))^{0.5}\right)$$

After the send, on line 315 we have reached the final fractional tree $\epsilon$.

Back in the left-hand thread, the second send is received in line 211, leading to the weakly-separated postcondition in line 212. In line 213 we "jump" label $\gamma$, and then in line 214 we use the known disjointness of $\gamma$ and $\delta$ to change the $\circledast$ to $*$. Finally in line 215 we apply the magic wand to reach the postcondition.

## 6 Conclusions and future work

We propose an extension of separation logic with fractional permissions [4] in order to reason about sharing over arbitrary regions of memory. We identify two fundamental logical principles that fail when the "weak" separating conjunction $\circledast$ is used in place of the usual "strong" $*$, the first being distribution of permissions — $A^\pi \circledast B^\pi \not\models (A \circledast B)^\pi$ — and the second being the re-combination of permission-divided formulas, $A^\pi \circledast A^\sigma \not\models A^{\pi \oplus \sigma}$. We avoid the former difficulty by *retaining* the strong $*$ in the formalism alongside $\circledast$, and the latter by using nominal *labels*, from hybrid logic, to record exact aliasing between read-only copies of a formula.

The main previous work addressing these issues, by Le and Hobor [24], uses a combination of permissions based on *tree shares* [17] and semantic side conditions on formulas to overcome the aforementioned problems. The *rely-guarantee* separation logic in [30] similarly restricts concurrent reasoning to structures described by precise formulas only. In contrast, our logic is a little more complex, but we can use permissions of any kind, and do not require side conditions. In addition, our use of labelling enables us to handle examples involving the transfer of data structures between concurrent threads.

On the other hand, we think it probable that the kind of examples we consider in this paper could also be proven by hand in at least some of the verification formalisms derived from CSL (e.g. [16, 27, 22]). For example, using the "concurrent abstract predicates" in [16], one can explicitly declare shared regions of memory in a fairly ad-hoc way. However, such program logics are typically very complicated and, we believe, quite unlikely to be amenable to automation.

We feel that the main appeal of the present work lies in its relative simplicity — we build on standard CSL with permissions and invoke only a modest amount of extra syntax — which bodes well for its potential automation (at least for simpler examples). In practical terms, an obvious way to proceed would be to develop a prototype verifier for concurrent programs based on our logic $\mathsf{SL}_{\mathsf{LP}}$. An important challenge in this area is to develop heuristics — e.g., for splitting, labelling and combining formulas — that work acceptably well in practice.

An even greater challenge is to move from *verifying* user-provided specifications to *inferring* them automatically, as is done e.g. by Facebook Infer. In separation logic, this crucially depends on solving the *biabduction* problem, which aims to discover "best fit" solutions for applications of the frame rule [11, 9]. In the CSL setting, a further problem seems to lie in deciding how applications of the concurrency rule should divide resources between threads.

Finally, automating the verification approach set out in this paper will likely necessitate restricting our full logic to some suitably tractable fragment, e.g. one analogous to the well-known *symbolic heaps* in standard separation logic (cf. [2, 15]). The identification of such tractable fragments is another important theoretical problem in this area. It is our hope that this paper will serve to stimulate interest in the automation of concurrent separation logic in particular, and permission-sensitive reasoning in general.

## References


1. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: Program Logics for Certified Compilers. Cambridge University Press, New York, NY, USA (2014)
2. Berdine, J., Calcagno, C., O'Hearn, P.: A decidable fragment of separation logic. In: Proceedings of FSTTCS-24. LNCS, vol. 3328, pp. 97–109. Springer (2004)
3. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press (2001)
4. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: Proceedings of POPL-32. pp. 59–70. ACM (2005)
5. Boyland, J.: Checking interference with fractional permissions. In: Proceedings of SAS-10. pp. 55–72. Springer (2003)
6. Brookes, S.: A semantics for concurrent separation logic. Theoretical Computer Science **375**(1–3), 227–270 (2007)
7. Brotherston, J.: Formalised inductive reasoning in the logic of bunched implications. In: Proceedings of SAS-14. pp. 87–103. Springer (2007)
8. Brotherston, J., Fuhs, C., Gorogiannis, N., Navarro Pérez, J.: A decision procedure for satisfiability in separation logic with inductive predicates. In: Proceedings of CSL-LICS. pp. 25:1–25:10. ACM (2014)
9. Brotherston, J., Gorogiannis, N., Kanovich, M.: Biabduction (and related problems) in array separation logic. In: Proceedings of CADE-26. pp. 472–490. Springer (2017)
10. Brotherston, J., Villard, J.: Parametric completeness for separation theories. In: Proceedings of POPL-41. pp. 453–464. ACM (2014)
11. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6) (December 2011)
12. Calcagno, C., O'Hearn, P., Yang, H.: Local action and abstract separation logic. In: Proceedings of LICS-22. pp. 366–378. IEEE Computer Society (2007)
13. Cao, Q., Wang, S., Hobor, A., Appel, A.W.: Proof pearl: Magic wand as frame (2019)
14. Costea, A., Chin, W., Qin, S., Craciun, F.: Automated modular verification for relaxed communication protocols. In: Proceedings of APLAS-16. pp. 284–305 (2018)
15. Demri, S., Lozes, E., Lugiez, D.: On symbolic heaps modulo permission theories. In: Proceedings of FSTTCS-37. pp. 25:1–25:13. Dagstuhl (2017)
16. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: Proceedings of ECOOP-24. pp. 504–528. Springer (2010)
17. Dockins, R., Hobor, A., Appel, A.W.: A fresh look at separation algebras and share accounting. In: Proceedings of APLAS. pp. 161–177. Springer (2009)
18. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Proceedings of ESOP-17. pp. 353–367 (2008)
19. Hobor, A., Gherghina, C.: Barriers in concurrent separation logic: Now with tool support! Logical Methods in Computer Science **8**(2) (2012)
20. Hobor, A., Villard, J.: The ramifications of sharing in data structures. In: Proceedings of POPL-40. pp. 523–536. ACM (2013)
21. Hóu, Z., Clouston, R., Goré, R., Tiu, A.: Proof search for propositional abstract separation logics via labelled sequents. In: Proceedings of POPL-41. pp. 465–476. ACM (2014)

22. Krebbers, R., Jung, R., Bizjak, A., Jourdan, J.H., Dreyer, D., Birkedal, L.: The essence of higher-order concurrent separation logic. In: Proceedings of ESOP-26. pp. 696–723. Springer (2017)
23. Larchey-Wendling, D., Galmiche, D.: Exploring the relation between intuitionistic BI and Boolean BI: An unexpected embedding. Mathematical Structures in Computer Science **19**, 1–66 (2009)
24. Le, X.B., Hobor, A.: Logical reasoning for disjoint permissions. In: Proceedings of ESOP-27. pp. 385–414. Springer (2018)
25. Lee, W., Park, S.: A proof system for separation logic with magic wand. In: Proceedings of POPL-41. pp. 477–490. ACM (2014)
26. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science **375**(1–3), 271–307 (2007)
27. Raad, A., Villard, J., Gardner, P.: Colosl: Concurrent local subjective logic. In: Proceedings of ESOP-24. pp. 710–735. Springer (2015)
28. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of LICS-17. pp. 55–74. IEEE Computer Society (2002)
29. Vafeiadis, V.: Concurrent separation logic and operational semantics. In: Proceedings of MFPS-27. pp. 335–351. Elsevier (2011)
30. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Proceedings of CONCUR-18. pp. 256–271. Springer (2007)
31. Villard, J., Lozes, É., Calcagno, C.: Tracking heaps that hop with heap-hop. In: Proceedings of TACAS-16. pp. 275–279. Springer (2010)
32. Yang, H., O'Hearn, P.: A semantic basis for local reasoning. In: Proceedings of FOSSACS-5. pp. 402–416. Springer (2002)