Proving Copyless Message Passing

Jules Villard[1] Étienne Lozes[1] Cristiano Calcagno[2]

[1] LSV, ENS Cachan, CNRS [2] Imperial College, London

Abstract. Handling concurrency using a shared memory and locks is tedious and error-prone. One solution is to use message passing instead. We study here a particular, contract-based flavor that makes the ownership transfer of messages explicit. In this case, ownership of the heap region representing the content of a message is lost upon sending, which can lead to efficient implementations. In this paper, we define a proof system for a concurrent imperative programming language implementing this idea and inspired by the Singularity OS. The proof system, for which we prove soundness, is an extension of separation logic, which has already been used successfully to study various ownership-oriented paradigms.

Introduction

Asynchronous message passing often suffers from two drawbacks: contents of messages have to be copied, and deadlocks can be tricky to avoid. However, if messages to-be live in the same address space, the first issue can be resolved by sending a mere pointer to the memory region where the message is stored instead of issuing a copy. This implementation is sound provided that the emitting thread loses ownership over the message, *i.e.* does not access it for reading or writing after emission.

The goal of this paper is to give a semantics and a proof theory for this way of programming. Our idealized programming language allows memory manipulation and asynchronous communications ruled by contracts, a basic form of session types, following the ideas of `Sing#`. Our proof system is based on separation logic [12], which has already been used to specify and prove various ownership-based paradigms [10,4]. Contracts play an essential role in this proof system: message invariants are associated to every contract's message, in the same spirit as resource invariants in concurrent separation logic [10]. Moreover, we show that they can ensure the absence of memory leaks when channels are closed.

To better illustrate copyless message passing, consider the following code snippet, where x, y can be thought of as buffers, and e, e' as the two endpoints of a channel:

```
(e,e') = open();
send(e,x);
y = receive(e');
close(e,e');
```

In a copying implementation, a whole copy of the buffer x would be allocated, and its address stored in y, whereas in copyless message passing, the whole code would be equivalent to `x = y`.

Our first contribution is the proof of soundness for our proof system: provable programs do not fault on memory accesses, are race free, and are contract obedient. However, and unlike for concurrent separation logic, it cannot entail the absence of memory leaks, due to the possibility of non-local leaks when channels are closed.

Finding a semantics that both establishes the soundness of our proof system and corresponds to the intended behaviour was challenging, because the standard semantics for separation logic is a local semantics, for which for instance the equivalence between the code above and `x = y` would not hold. Our second contribution is to propose a new approach for defining such a semantics. Indeed, we define a *local* semantics based on abstract separation logic [5], for which our proof system is sound, and then restrict it to a *global* semantics, also sound by restriction. However, neither the local nor the global semantics reflect the intended semantics faithfully.

Our third contribution is to state and prove a more general result that entails the validity of the pointer-passing implementation and the absence of memory leaks. This result, which we christen the transfers erasure property, relates the global semantics to the intended semantics in a non-trivial way, and allows to state that programs p for which the Hoare triple $\{\mathsf{emp}\}\ p\ \{\mathsf{emp}\}$ is valid are leak free under some conditions on the contracts.

We first introduce the language and its main features by a small motivating example. We then present the programming language and our proof system in Sec. 2, and demonstrate how to prove the example. Sec. 3 gives an overview of the main ingredients of our semantics. We develop the semantics in more details in Sec. 4, leading to a soundness result for our logic. Sec. 5 is devoted to the transfers erasure property.

Related work The Singularity operating system [6] is a prominent application of contract-based copyless message passing ideas. It can safely run processes sharing a unique address space without memory protection. Executables are written in the `Sing#` programming language, which supports (copyless) message passing primitives. Ownership violations are detected at compile-time using static analysis techniques, and communications are ruled by contracts. Our work can be seen as an abstract model of `Sing#`, though some differences between the two are worth noting: we chose to be able to detect memory leaks, while `Sing#` is equipped with a garbage collector, and we support complete mobility of channels, similar to π-calculus, whereas `Sing#` provides internal mobility only. Finally, as our language is not full-fledged, we did not provide mechanisms for error handling, for example when one endpoint is abruptly closed.

Concurrent Separation Logic [10] and the logic of Gotsman & al. for locks in the heap [7] inspired our work. While the former cannot handle an unbounded number of resources, it would surely have been possible to encode message passing commands in the toy programming language of the latter. However, contracts seem of such a different nature that it appeared simpler to take message passing instructions as primitive. More importantly, the local semantics used in these works is an over-approximation of the intended semantics, as the exchange of shared resources involves a possible non-deterministic change of the resource content provided it still respects some invariant. The transfers erasure property cannot be established in these approaches.

Contracts may be viewed as session types [13]. However, the approach is different in this work, as our main concern with contracts is how they can help us to prove that

channels do not leak memory, and not how they can prove the absence of communication errors (although this certainly is an interesting topic for future work). Session types were also used on top of Java in SessionJ [9], but this does not address the problem of copyless message passing.

Pym and Tofts [11], and O'Hearn and Hoare [8] have defined two other logics for resource aware message passing programs. However, their respective models differ significantly from ours as they are based on process algebras, and are not centered around memory management.

1 Programming language

1.1 Contracts

Contracts describe the behavior of channels. A channel is asynchronous, bi-directional, and has two endpoints, distinguished for ease of reference: the serving endpoint and the client endpoint. Contracts are state machines describing what sends (`!`) and receives (`?`) are allowed in a given state. They are written from the server's point of view, the client's one being dual. Each message sent over the channel is described by a message identifier. Moreover, each message identifier is annotated with an invariant (between brackets) for proofs' purpose. These invariants are separation logic formulas, and replace `Sing#` messages' types. Their syntax and purpose will be explained in the next sections.

The contract `C` below describes the protocol implemented by our example. It has three states, three transitions, and three messages may trigger these transitions.

```
contract C {
  message ack       [emp]
  message cell      [val ↦ X]
  message close_me  [src ↦^ep (C{end}, −) ∧ src = val]

  initial state transfer { !cell --> wait_ack;
                           !close_me --> end; }
  state wait_ack { ?ack --> transfer; }
  final state end {} }
```

`ack` is a message used for synchronization purpose only, whereas `cell` and `close_me` respectively carry (the addresses of) a list's head and an endpoint. On a channel following `C`, the serving endpoint would be able to perform as many sequences of sending a memory cell and then waiting for an acknowledgment as it wishes, and will eventually send a `close_me` message to go to the final state `end`. This protocol can be used to send a linked list over the channel until it is empty, and finally request a closing of the channel, as we will see next.

1.2 Sending a list cell by cell

The imperative programming language we use features standard variable and memory manipulation. We moreover use `send(m, e, x)` to send message m with value x over endpoint e and `x = receive(m, f)` to retrieve this value through f, provided it is the other end of the channel (*i.e.* f is the *peer* of e). Intuitively, `send` and `receive` are

asynchronous communications, and act as enqueuing and dequeuing over one of the two queues that are shared by two coupled endpoints (one queue for each direction). Endpoints are allocated on the heap upon channel creation (`(e,f)=open(C)`) and closed together (`close(e,f)`). This differs from most implementations, where endpoints can be closed independently, but this case an implicit message is sent to notify the closing of one of the endpoint to the other; in our setting, such a message has to be sent explicitly.

Let us now give a program implementing contract `C`. The serving endpoint e is held by the `putter` program, which communicates with `getter`. The program is given a list starting at the address x that it sends cell by cell over e. `getter` disposes the cells one by one, and when the list becomes empty, `putter` sends its endpoint over itself so that `getter` may close the channel. Comments (lines starting with `//` and annotations between brackets) are elements of the proof and will be explained later.

```
putter(e,x) [e, x ⊩ e ↦^ep (C{transfer}, X) ∗ list(x)] {
  local t;
  while (x != 0) {
    // e, x, t ⊩ x ↦ Y ∗ list(Y) ∗ e ↦^ep (C{transfer}, X)
    t = *x;
    send(cell, e, x);
    // e, x, t ⊩ list(t) ∗ e ↦^ep (C{wait_ack}, X)
    x = t;
    receive(ack, e); }
  // e, x, t ⊩ e ↦^ep (C{transfer}, X)
  send(close_me, e, e); } [e, x ⊩ emp]

getter(f) [f ⊩ f ↦^ep (C̄{transfer}, Y)] {
  local x, e = 0;
  // 0 = x, e, f
  while (e == 0) {
    // O ⊩ f ↦^ep (C̄{transfer}, Y) ∗ e = 0
    switch receive {
    x = receive(cell, f): {
      // O ⊩ f ↦^ep (C̄{wait_ack}, Y) ∗ e = 0 ∗ x ↦ −
      free(x)
      // O ⊩ f ↦^ep (C̄{wait_ack}, Y) ∗ e = 0
      send(ack, f); }
    e = receive(close_me, f): {} }}
  // O ⊩ e ↦^ep (C{end}, f) ∗ f ↦^ep (C̄{end}, e)
  close(e, f); } [f ⊩ emp]

main() [x ⊩ list(x)] {
  local e,f;
  (e,f) = open(C);
  // x, e, f ⊩ list(x) ∗ e ↦^ep (C{transfer}, f) ∗ f ↦^ep (C̄{transfer}, e)
  putter(e,x); || getter(f); } [x ⊩ emp]
```

2 A separation logic for copyless message passing

2.1 Syntax of programs

We assume infinite sets $Var = \{e, f, x, y, \dots\}$, $Loc = \{l, \dots\}$, $Endpoint = \{\varepsilon, \dots\}$, $MsgId = \{m, \dots\}$, $State = \{a, b, \dots\}$ and $Val = \{v, \dots\}$ of respectively variables,

memory locations, endpoints, message identifiers, contracts' states and values. All sets but values are pairwise disjoint, and $Loc \uplus Endpoint \uplus \{0\} \subseteq Val$. The grammar of expressions, boolean expressions, atomic commands and programs is as follows:

$$E ::= x \in Var \mid v \in Val \qquad B ::= E = E \mid B \text{ and } B \mid \text{not } B$$

$$c ::= \text{assume(B)} \mid \text{x = E} \mid \text{x = new()} \mid *\text{E = E} \mid \text{x = } *\text{E} \mid \text{free (E)} \\ \mid \text{(e, f) = open(C)} \mid \text{close (E,E)} \mid \text{send(m, E, E)} \mid \text{x = receive (m, E)}$$

$$p ::= c \mid p;\ p \mid p \parallel p \mid p + p \mid p^* \mid \text{local x in } p$$

`assume(B)` blocks unless B holds, and does nothing otherwise. Compound commands are standard and are in order sequential and parallel composition, non-deterministic choice, Kleene iteration and local variable creation. `switch receive` is defined as a non-deterministic choice of the `{x=receive(m,E); p}` for every `x = receive(m,E): p` of its body. We leave the similar definitions of `while` loops and `if` statements to the attention of the reader. In our example of Sec. 1.2, subroutines `putter` and `getter` should actually be inlined to fit our model, as it does not feature procedure calls. We write $v(E)$ the set of variables that appear in expression E.

Contracts A contract is an edge-labeled oriented graph. Vertices are called *states*, and every contract C distinguishes an initial state $\mathrm{init}(C)$ and a set of final states $\mathrm{final}(C)$. Labels are either send label $!m$ or receive label $?m$, where m is a message identifier. We write $\bar{C}$ for the dual of contract C, *i.e.* C where ! and ? are swapped.

A contract specification is given by a map $m \mapsto I_m$ from message identifiers to precise separation logic formulas (to be defined soon). I_m is called the invariant of the message. Only special variables `val` and `src` can appear free in I_m.

2.2 Syntax of the logic

We assume an extra infinite set $LVar = \{X, Y, \dots\}$ of logical variables distinct from the program's variables. We extend the grammar of expressions to allow them to contain logical variables. The assertion language is then as follows:

$$\begin{array}{rll} A ::= & \mathsf{emp}_s \mid \mathsf{own}(x) \mid E = E & \text{stack predicates} \\ & \mid \mathsf{emp}_h \mid \mathsf{emp}_{ep} \mid E \mapsto E \mid E \overset{ep}{\mapsto} (C\{a\}, E) & \text{heaps predicates} \\ & \mid \neg A \mid A \wedge A \mid \exists X.\, A \mid A * A \mid A \mathrel{-\!\!*} A & \text{connectives} \end{array}$$

All predicates and connectives are standard, except emp_{ep} and $E \overset{ep}{\mapsto} (C\{a\}, E)$. Before defining the semantics of the logic, let us define some useful shorthands.

We will write emp for $\mathsf{emp}_h \wedge \mathsf{emp}_{ep}$ and, for instance, $E \mapsto -$ for $\exists X.\, E \mapsto X$. In this work, we use variables as resources [2] without permissions for simplicity, thus forbidding concurrent reads. When $O = x_1, \dots, x_n$, we will write, as usual, $O \Vdash A$ as a shorthand for $(\mathsf{own}(x_1) * \dots * \mathsf{own}(x_n)) \wedge A$. Moreover, to avoid cumbersome notations in formulas, we will sometimes allow reads to the same variable in the stack in two disjoint states, *i.e.* have formulas of the form $x \Vdash A(x) * B(x)$. They should be understood as $x \Vdash \exists X.\, x = X \wedge (A(X) * B(X))$.

2.3 Basic memory model

Formulas are interpreted over a subset Σ^{wf} of the set Σ of basic memory pre-states (s, h, k) defined by:

$$\Sigma \triangleq \mathit{Stack} \times \mathit{CHeap} \times \mathit{EHeap} \qquad \mathit{Stack} \triangleq \mathit{Var} \rightharpoonup \mathit{Val} \qquad \mathit{CHeap} \triangleq \mathit{Loc} \rightharpoonup \mathit{Val}$$

$$\mathit{EHeap} \triangleq \mathit{Endpoint} \rightharpoonup \mathit{Contract} \times \mathit{State} \times \mathit{Endpoint}$$

It is equipped with a composition law $\circ$ of a separation algebra (see Sec. 3.1) defined as the disjoint union $\uplus$ of each of the components of the pre-states: $(s, h, k) \circ (s', h', k') \triangleq (s \uplus s', h \uplus h', k \uplus k')$. *Stack* and *CHeap* (cell heap) are standard, and *EHeap* (endpoint heap) works in the same way as *CHeap* but is used to represent endpoints.

We define *memory states* Σ^{wf} as the elements of Σ that satisfy the axioms

$$\begin{aligned}
\textbf{Dual}\quad & k(\varepsilon) = (C, a, \varepsilon') \;\&\; k(\varepsilon') = (C', b, \varepsilon'') \Rightarrow \varepsilon'' = \varepsilon \;\&\; C' = \bar{C} \\
\textbf{Irreflexive}\quad & k(\varepsilon) = (-, -, \varepsilon') \Rightarrow \varepsilon \neq \varepsilon' \\
\textbf{Injective}\quad & k(\varepsilon_1) = (-, -, \varepsilon_1') \;\&\; k(\varepsilon_2) = (-, -, \varepsilon_2') \;\&\; \varepsilon_1 \neq \varepsilon_2 \Rightarrow \varepsilon_1' \neq \varepsilon_2'
\end{aligned}$$

We restrict $\circ$ to a new operation $\bullet$ on memory states defined only when $\sigma \circ \sigma' \in \Sigma^{\mathsf{wf}}$. We will write $\sigma \sharp \sigma'$ when this is the case. Let us now give the satisfaction relation $\vDash$ between states in Σ^{wf} and formulas. We write $[\![x]\!]s$ to denote $s(x)$ if $x \in \mathit{dom}(s)$, and $[\![v]\!]s$ denotes v.

$$\begin{aligned}
(s,h,k) &\vDash E_1 = E_2 && \text{iff } v(E_1, E_2) \subseteq \mathit{dom}(s) \;\&\; [\![E_1]\!]s = [\![E_2]\!]s \\
(s,h,k) &\vDash \mathsf{emp}_{\spadesuit} && \text{iff } \mathit{dom}(\spadesuit) = \emptyset \quad (\spadesuit \in \{s, h, k\}) \\
(s,h,k) &\vDash \mathsf{own}(x) && \text{iff } \mathit{dom}(s) = \{x\} \\
(s,h,k) &\vDash E_1 \mapsto E_2 && \text{iff } v(E_1, E_2) \subseteq \mathit{dom}(s) \;\&\; \mathit{dom}(h) = \{[\![E_1]\!]s\} \\
& && \quad \&\; \mathit{dom}(k) = \emptyset \;\&\; h([\![E_1]\!]s) = [\![E_2]\!]s \\
(s,h,k) &\vDash E_1 \overset{ep}{\mapsto} (C\{a\}, E_2) && \text{iff } v(E_1, E_2) \subseteq \mathit{dom}(s) \;\&\; \mathit{dom}(k) = \{[\![E_1]\!]s\} \\
& && \quad \&\; \mathit{dom}(h) = \emptyset \;\&\; k([\![E_1]\!]s) = (C, a, [\![E_2]\!]s) \\
\sigma &\vDash \neg A && \text{iff } \sigma \nvDash A \\
\sigma &\vDash A_1 \wedge A_2 && \text{iff } \sigma \vDash A_1 \;\&\; \sigma \vDash A_2 \\
\sigma &\vDash \exists X.\, A && \text{iff } \exists v \in \mathit{Val}.\, \sigma \vDash A[X \leftarrow v] \\
\sigma &\vDash A_1 * A_2 && \text{iff } \exists \sigma_1, \sigma_2.\, \sigma = \sigma_1 \bullet \sigma_2 \;\&\; \sigma_1 \vDash A_1 \;\&\; \sigma_2 \vDash A_2 \\
\sigma &\vDash A \mathrel{-\!\!*} B && \text{iff } \forall \sigma' \sharp \sigma.\, \sigma' \vDash A \text{ implies } \sigma \bullet \sigma' \vDash B
\end{aligned}$$

2.4 Proof system

Our proof system is based on the framework of abstract separation logic. We extend the rules of separation logic (frame rule, composition rules and the standard small axioms for all pointer instructions) with four new small axioms for channel instructions. We abbreviate $I_m[\mathtt{src} \leftarrow E_1, \mathtt{val} \leftarrow E_2]$ as $I_m(E_1, E_2)$. Figure 1 presents all the rules. Among these four new small axioms, the one for `send` deserves a special attention, as we can derive two different small axioms from it: $\{O \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon) * I_m(E, F)\}$ send(m,E,F) $\{O \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon)\}$ that accounts for the most standard sending (taking $A = E \overset{ep}{\mapsto} (C\{b\}, \varepsilon)$), and sending the endpoint over itself is accounted by $\{O \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon) * (E \overset{ep}{\mapsto} (C\{b\}, \varepsilon) \mathrel{-\!\!*} I_m(E, F))\}$ send(m,E,F) $\{O \Vdash \mathsf{emp}\}$ (taking $A = \mathsf{emp}$). We will write $\vdash \{A\}$ p $\{B\}$ when this triple is derivable.

Figure 1 Proof System Rules

$$\frac{\boldsymbol{x} = v(B)}{\{\boldsymbol{x} \Vdash \boldsymbol{x} = \boldsymbol{v}\}\ \text{assume(B)}\ \{\boldsymbol{x} \Vdash \boldsymbol{x} = \boldsymbol{v} \wedge B\}} \qquad \{x, O \Vdash E = v \wedge \mathsf{emp}\}\ \text{x = E}\ \{x, O \Vdash x = v \wedge \mathsf{emp}\}$$

$$\{x \Vdash \mathsf{emp}\}\ \text{x = new()}\ \{x \Vdash x \mapsto -\} \qquad \{O \Vdash E \mapsto - \wedge F = v\}\ *\text{E = F}\ \{O \Vdash E \mapsto v\}$$

$$\{x, O \Vdash E = v \wedge v \mapsto v'\}\ \text{x = *E}\ \{x, O \Vdash x = v' \wedge v \mapsto v'\} \qquad \{O \Vdash E \mapsto -\}\ \text{free (E)}\ \{O \Vdash \mathsf{emp}\}$$

$$\frac{i = \mathrm{init}(C)}{\{e, f \Vdash \mathsf{emp}\}\ \text{(e,f) = open(C)}\ \{e, f \Vdash e \overset{ep}{\mapsto} (C\{i\}, f) * f \overset{ep}{\mapsto} (\bar{C}\{i\}, e)\}}$$

$$\frac{a \in \mathrm{final}(C)}{\{O \Vdash E \overset{ep}{\mapsto} (C\{a\}, E') * E' \overset{ep}{\mapsto} (\bar{C}\{a\}, E)\}\ \text{close (E,E')}\ \{O \Vdash \mathsf{emp}\}}$$

$$\frac{a \xrightarrow{!m} b \in C}{\{O \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon) * (E \overset{ep}{\mapsto} (C\{b\}, \varepsilon) \mathrel{-\!\!*} (I_m(E, F) * A))\}\ \text{send(m,E,F)}\ \{O \Vdash A\}}$$

$$\frac{a \xrightarrow{?m} b \in C}{\{O, x \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon)\}\ \text{x = receive (m,E)}\ \{O, x \Vdash E \overset{ep}{\mapsto} (C\{b\}, \varepsilon) * I_m(\varepsilon, x)\}} \qquad \frac{\{A\}\ p\ \{B\}}{\{A * F\}\ p\ \{B * F\}}$$

$$\frac{A' \Rightarrow A \qquad \{A\}\ p\ \{B\} \qquad B \Rightarrow B'}{\{A'\}\ p\ \{B'\}} \qquad \frac{\{A_i\}\ p\ \{B_i\} \quad \text{all i in } I}{\{\bigsqcap_{i \in I} A_i\}\ p\ \{\bigsqcap_{i \in I} B_i\}} \qquad \frac{\{A_i\}\ p\ \{B_i\} \quad \text{all i in } I}{\{\bigsqcup_{i \in I} A_i\}\ p\ \{\bigsqcup_{i \in I} B_i\}}$$

$$\frac{\{A\}\ p\ \{B\} \qquad \{A'\}\ p'\ \{B'\}}{\{A * A'\}\ p \parallel p'\ \{B * B'\}} \qquad \frac{\{A\}\ p\ \{A'\} \qquad \{A'\}\ p'\ \{B\}}{\{A\}\ p; p'\ \{B\}} \qquad \frac{\{A\}\ p\ \{B\} \qquad \{A\}\ p'\ \{B\}}{\{A\}\ p + p'\ \{B\}}$$

$$\frac{\{I\}\ p\ \{I\}}{\{I\}\ p^*\ \{I\}} \qquad \frac{\{\mathsf{own}(z) * A\}\ p[x \leftarrow z]\ \{\mathsf{own}(z) * B\}}{\{A\}\ \text{local } x \text{ in } p\ \{B\}}\ z \text{ fresh}$$

2.5 Back to the example

We now highlight some steps of the proof that the program p presented at Sec. 1.2 satisfies the Hoare triple $\{x \Vdash \mathrm{list}(x)\}\ p\ \{x \Vdash \mathsf{emp}\}$. Bracketed formulas are used to denote the pre and post-condition of a program. We start with the precondition $x \Vdash \mathrm{list}(x)$, where $\mathrm{list}(x)$ is the inductive list predicate verifying emp if $x = 0$ and $\exists X.\, x \mapsto X * \mathrm{list}(X)$ otherwise. Before entering the parallel composition, we obtain $x, e, f \Vdash \mathrm{list}(x) * e \overset{ep}{\mapsto} (C\{\texttt{transfer}\}, f) * f \overset{ep}{\mapsto} (\bar{C}\{\texttt{transfer}\}, e)$. To apply the rule for parallel composition, we have to split the state into two. `putter` will get resources $e, x \Vdash e \overset{ep}{\mapsto} (C\{\texttt{transfer}\}, -) * \mathrm{list}(x)$ and `getter` $f \Vdash f \overset{ep}{\mapsto} (\bar{C}\{\texttt{transfer}\}, -)$. The next important step is after the loop, at line 9; we are left with just the endpoint e, which we send in a `close_me` message. According to $I_{\texttt{close\_me}}$ and the rule of send with $A = \mathsf{emp}$, the post-condition of `putter` is thus $e, x \Vdash \mathsf{emp}$.

The proof of the `getter` program follows the same lines. Crucially, after receiving the `close_me` message, we can deduce that we have received the *peer* of f thanks to **Dual** and the use of the `src` variable in $I_{\texttt{close\_me}}$. This allows the CLOSE rule to fire

up. At the end of the parallel composition, we thus obtain empty heaps $x \Vdash \mathsf{emp}$ which concludes the proof.

3 Soundness

We now turn to proving that the proof system is sound and giving an accurate semantics for programs. As the soundness of concurrent separation logic itself has proven hard to establish in the past [3], we base our work on abstract separation logic, which allows us to deduce the soundness of our proof system from the sole soundness of its axioms. But this only resolves half of our concerns, for concurrent separation logic is not fit to describe synchrony issues, for example that sends must happen before receives, nor does it entail a pointer passing semantics, without transfers (the transfers erasure property). In this section, we explain how to remedy this by extending the basic memory model.

As even an informal presentation of our semantics relies heavily on abstract separation logic [5], we begin this section by a short introduction to this framework.

3.1 Abstract separation logic in a nutshell

A *separation algebra* is a cancellative, partial, commutative monoid $(\Sigma, \bullet, u)$ where cancellative means that the partial function $\sigma \bullet (\cdot) : \Sigma \rightharpoonup \Sigma$ is injective; one may write either $\sigma_1 \sharp \sigma_2$ or $\sigma_1 \bot \sigma_2$ when $\sigma_1 \bullet \sigma_2$ is defined, $\sigma \preceq \sigma'$ if there is σ_1 such that $\sigma' = \sigma \bullet \sigma_1$, and denote the unique such σ_1 by $\sigma' - \sigma$ when it exists. $(\mathcal{P}(\Sigma)^\top, \sqsubseteq)$ denotes the powerset of Σ ordered by inclusion, extended with a greatest element $\top$. The operator $*$ defined by $A * B = \{\sigma_0 \bullet \sigma_1 \mid \sigma_0 \sharp \sigma_1 \,\&\, \sigma_0 \in A \,\&\, \sigma_1 \in B\}$ if $A, B \neq \top$, $\top$ otherwise, defines a commutative ordered monoid $(\mathcal{P}(\Sigma)^\top, *, \emptyset, \sqsubseteq)$. A property A is *precise* if for all σ, there is at most one $\sigma' \preceq \sigma$ in A.

We will later define the semantics of all atomic commands as local functions. A *local function* $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ is a total function such that for all $\sigma, \sigma' \in \Sigma$, if $\sigma \sharp \sigma'$ then $f(\sigma \bullet \sigma') \sqsubseteq \{\sigma\} * f(\sigma')$. $f \sqsubseteq g$ denotes the pointwise order on local functions. Composition $f; g$ of local functions is performed using the obvious lifting of g to $\mathcal{P}(\Sigma)^\top$: $(f; g)(\sigma) \triangleq \bigsqcup\{g(\sigma') \mid \sigma' \in f(\sigma)\}$ or $\top$ if $f(\sigma) = \top$. A *specification* ϕ is a set of pairs (A, B) in $\mathcal{P}(\Sigma)$. We write $f \vDash \phi$ and say that f satisfies ϕ when $f(A) \sqsubseteq B$ for every $(A, B) \in \phi$. The best local action of ϕ is defined by $bla[\phi](\sigma) = \bigsqcap_{\sigma' \preceq \sigma, \sigma' \in A, (A,B) \in \phi} \{\sigma - \sigma'\} * B$. It is local, satisfies its specification, and is the greatest such local function for the pointwise order on local functions [5].

Lemma 1. *Basic memory states* $(\Sigma^{\mathsf{wf}}, \bullet, u)$ *and pre-states* $(\Sigma, \circ, u)$*, where* $u = (\emptyset, \emptyset, \emptyset)$*, are separation algebras.*

A simple way to obtain a soundness result for our proof system would thus be to define the semantics of all atomic commands as the best local actions of their specifications. Using the trace semantics we will present soon, all proof rules would be sound. As we will explain now, this would lead to a very coarse semantics without synchronization that over-approximates the communications. In particular, as mentionned in the introduction, sending a message with value x and immediately retrieving it in y would not be equivalent to simply assigning x to y.

3.2 Trace semantics and global semantics

Syntactic traces Let us define the traces $T(p)$ of a program p as a set of sequences of *actions* $\alpha \in \{c, \mathrm{norace}(c_1, c_2), \mathrm{n}_x, \mathrm{d}_x\}$ for all commands c, c_1, c_2, following the original approach of abstract separation logic extended with the treatment of local variables: n_x allocates $x \in \mathit{Var}$ on the stack and d_x disposes it.

$$T(\alpha) = \{\alpha\} \qquad T(p_1 + p_2) = T(p_1) \cup T(p_2) \qquad T(p^*) = (T(p))^*$$

$$T(p_1; p_2) = \{tr_1; tr_2 \mid tr_i \in T(p_i)\} \qquad T(p_1 \parallel p_2) = \{tr_1 \mathbin{zip} tr_2 \mid tr_i \in T(p_i)\}$$

$$T(\text{ local x in } p) = \{\mathrm{n}_z; T(p[x{\leftarrow}z]); \mathrm{d}_z \mid z \text{ fresh in } p\}$$

Parallel composition is treated as a syntactic interleaving of commands. We force all racy programs to fault by placing $\mathrm{norace}(c_1, c_2)$ each time c_1 and c_2 may be executed simultaneously. This command will check that c_1 and c_2 can execute on disjoint portions of the state. zip is thus defined by $\varepsilon \mathbin{zip} tr = tr \mathbin{zip} \varepsilon = tr$ in the base case, and by $(c_1; tr_1) \mathbin{zip} (c_2; tr_2) = \mathrm{norace}(c_1, c_2); ((c_1; (tr_1 \mathbin{zip} (c_2; tr_2))) \cup (c_2; (c_1; tr_1) \mathbin{zip} tr_2))$.

Semantics The denotational semantics of traces is the composition of the interpretation of atomic actions: $[\![\alpha]\!] = (\!|\alpha|\!)$ and $[\![tr_1; tr_2]\!] = [\![tr_1]\!]; [\![tr_2]\!]$. This assumes that a semantics $(\!|c|\!)$ is defined for all primitive commands, which we will give later. The semantics of $\mathrm{norace}(c_1, c_2)$ is the local function $\mathit{norace}((\!|c_1|\!), (\!|c_2|\!))$, defined by

$$\mathit{norace}(f, g)(\hat{\sigma}) \triangleq \begin{cases} \{\hat{\sigma}\} & \text{if } \exists \hat{\sigma}_f, \hat{\sigma}_g.\, \hat{\sigma}_f \bullet \hat{\sigma}_g = \hat{\sigma} \;\&\; f(\hat{\sigma}_f) \neq \top \;\&\; g(\hat{\sigma}_g) \neq \top \\ \top & \text{otherwise} \end{cases}$$

Finally, the semantics of stack bookkeeping actions n_x and d_x are defined as the best local actions $(\!|\mathrm{n}_x|\!) \triangleq n_x \triangleq \mathit{bla}[\mathsf{emp}_s, \mathsf{own}(x)]$ and $(\!|\mathrm{d}_x|\!) \triangleq d_x \triangleq \mathit{bla}[\mathsf{own}(x), \mathsf{emp}_s]$.

Following this approach is essential for deriving easily the soundness of the parallel rule: $[\![p \parallel p']\!](\sigma \bullet \sigma') \sqsubseteq [\![p]\!](\sigma) * [\![p']\!](\sigma')$. The downside is that parallel threads have to work on disjoint memory states, hence receiving a message cannot be blocking and must be non-deterministic. This poses two challenges: how to synchronize concurrent actions and how to model inter-threads communication.

Successive semantics Our approach here relies on three successive refinements of the semantics defined by the small axioms of our proof system.

The first one enriches the memory model and the communication primitives so that, at any point in the execution, the history of all past communications, including the contents of all messages, can be observed in the resulting states. This is covered in Sec. 4. However, this semantics is still local so we cannot link the histories of two corresponding endpoints yet, as one of them may reside outside of the current heap. Moreover, `send` and `receive` still act respectively as a disposal and a non-deterministic creation of the message's contents.

The next step is thus to consider programs as wholes. In this case, we can observe the whole state at every point of the execution. In particular, both endpoints of every opened channel will always be present. We may now restrict the states produced by the commands to *legal* ones, *i.e.* states where receives have happened after the corresponding send, and where contents of sent and received messages match (intuitions about

how this will be performed are introduced in the next two subsections). This will automatically restrict traces in the same way: a receive preceding the corresponding send will produce an empty set of legal states. This is also how `assume(B)` works: it blocks executions where B does not hold.

The third and final semantics is the same as the global one, except that the communications are not loggued, thus achieving a pure pointer-passing semantics.

We will explain now what information histories will have to contain in the h.p. model. The formal definition of legal states can be found in Sec. 5.

3.3 Synchronization

Concurrent separation logic uses critical sections to synchronize and communicate between processes. In this case, the synchronization may be performed at the syntactic trace level, by considering only *well-formed* traces, in which two critical sections over the same resource are never interleaved [5]. This syntactic synchronization is possible because resources are not part of the expression language, and determining whether two critical sections refer to the same resource can be done just by looking at the resource identifier. This is not the case for channel communication, as retrieving which endpoint is used for sending or receiving involves evaluating an expression, which cannot be done at the trace level.

Instead, we rule out ill-synchronized traces at the semantics level by making them block when executed. To achieve this, we must add information to the endpoints in the model, namely how many messages have been sent and how many have been received on this endpoint (see Sec. 3.5), and modify `send` and `receive` to increment these counters. Legal states will thus be such that any endpoint ε must have received less than what its peer has sent. This ensures that traces where a `receive` happens when there is no pending message inside the channel will block.

3.4 Communication

In concurrent separation logic, communication is achieved by passing pieces of states around using conditional critical regions: acquiring a shared resource is modeled by an allocation (roughly, $(\!|\text{acquire } r|\!)(\sigma) = \{\sigma\} * I_r$), and releasing it is modeled by deallocation of the part of memory corresponding to the invariant. Acquiring a resource is thus non-deterministic, as the resource r may be acquired in any state satisfying the resource invariant I_r, and not the state in which it was left after the last release. The local semantics of `receive` suffer from the same caveat.

The semantics to which we aspire should be more precise and ensure that the contents of what is received match what was sent. For this purpose, we have chosen to "log" a copy of the message contents that is sent or non-deterministically received in the thread-local heap. To describe "logging", we enrich the model with *timestamps*: each cell and endpoint is tagged with a timestamp $\tau \in \mathcal{T}$ and a direction $\dagger \in \{?, !\}$. We will note $[\tau^\dagger]\hat{\sigma}$ for the memory state formed of a single log using timestamp τ. Sending $\hat{\sigma}$ will deallocate it and allocate the log $[\tau^!]\hat{\sigma}$, whereas the corresponding `receive` will allocate $\hat{\sigma}' \bullet [\tau^?]\hat{\sigma}'$ for the same timestamp τ and some guessed $\hat{\sigma}'$. Then, the memory

state $[\tau^!]\hat{\sigma} \bullet [\tau^?]\hat{\sigma}'$ will be declared legal if and only if $\hat{\sigma} = \hat{\sigma}'$, which will ensure the coherence of communications in the global semantics.

Moreover, we have to provide a mechanism for choosing which timestamp should be used for logging for each message, and the endpoint's owner that will issue a message should locally choose its timestamp. We thus attach to every channel a pair of *histories* $(\ell_!, \ell_?)$ where $\ell_!, \ell_?$ contain a list of the successive (distinct) timestamps at which the messages respectively sent and received (from the serving endpoint point of view) will have to be logged. Both endpoints will be equipped with these histories upon creation (the client endpoint will be equipped with the dual pair $(\ell_?, \ell_!)$) in order to log the same message with the same timestamp when it is sent and received.

Finally, histories can also be used to check that the value and message identifier of a message is the same on both endpoints involved. This could have been part of the logging mechanism, but it has turned out to be simpler to consider it apart.

3.5 Refined model

Adding up what has been informally described above, we obtain the following *history preserving memory model* $\hat{\Sigma}$ defined from histories $\mathit{Hist} \triangleq (\mathit{MsgId} \times \mathit{Val} \times \mathcal{T})^\omega$:

$$\begin{aligned} \hat{\Sigma} &\triangleq \mathit{Stack} \times \hat{\mathit{CHeap}} \times \hat{\mathit{EHeap}} \\ \hat{\mathit{CHeap}} &\triangleq \mathit{Loc} \times \mathcal{T}^{\{?,!\}} \rightharpoonup \mathit{Val} \\ \hat{\mathit{EHeap}} &\triangleq \mathit{Endpoint} \times \mathcal{T}^{\{?,!\}} \rightharpoonup \mathit{Contract} \times \mathit{State} \times \mathit{Endpoint} \times \mathbb{N}^2 \times \mathit{Hist}^2 \end{aligned}$$

Timestamps τ form an infinite set $\mathcal{T}$, disjoint from previously defined sets, from which we define *polarized* timestamps $\mathcal{T}^{\{?,!\}} \triangleq (\mathcal{T} \times \{?,!\}) + \{\mathbf{now}\}$. We extend Val to contain $\mathcal{T}$. For simplicity, we may write $\mathbf{now}^!$ and $\mathbf{now}^?$ for $\mathbf{now}$.

$(\hat{\Sigma}, \circ, u)$ defines a separation algebra where $\circ$ denotes disjoint union of (tuples of) partial functions. To distinguish heaps of the basic and h.p. model, we adopt a hat notation, and let $\hat{h}, \hat{k}, \ldots$ range over $\hat{\mathit{CHeap}}$, $\hat{\mathit{EHeap}}$.

We define the projection $\mathrm{now} : \hat{\Sigma} \to \Sigma$ which associates to a state $\hat{\sigma} = (s, \hat{h}, \hat{k}) \in \hat{\Sigma}$ the state $\sigma = (s, h, k)$ where $h = \hat{h}(\cdot, \mathbf{now})$ and k is $\hat{k}(\cdot, \mathbf{now})$ where histories and counters have been erased. We can now define what it means for a program executing on h.p. states to satisfy a Hoare triple.

Definition 1 (Semantic Hoare Triple).

- *If* $A, B \in \mathcal{P}(\Sigma)$ *and* $f : \hat{\Sigma} \to \mathcal{P}(\hat{\Sigma})^\top$*, we write* $\langle\langle A \rangle\rangle\ f\ \langle\langle B \rangle\rangle$ *iff* $\forall \hat{\sigma}.\, \mathrm{now}(\hat{\sigma}) \in A$ *implies* $\mathrm{now}(f(\hat{\sigma})) \sqsubseteq B$.
- *We write* $\vDash \{A\}\ p\ \{B\}$ *iff* $\forall tr \in T(p).\, \langle\langle A \rangle\rangle\ [\![tr]\!]\ \langle\langle B \rangle\rangle$.

4 Semantics of programs

4.1 Refined assertions

We now show how to interpret the logic in the refined model, so as to give a semantics of commands from logical specifications and state the soundness theorem later on.

We let $ts(\ell)$ denote the set of timestamps that appear in ℓ. If ℓ is a history list and i is an integer, $\ell[i]$ represents the i^{th} item of ℓ. We write $logs(\hat{\sigma})$ to denote the set of polarized timestamps that appear in $\hat{\sigma}$, *i.e.* $\mathrm{snd}(dom(\hat{h})) \cup \mathrm{snd}(dom(\hat{k}))$.

If $\hat{\sigma} = (s, \hat{h}, \hat{k})$, we write $\hat{\sigma}\downarrow_{\tau^\dagger}$ (resp. $\hat{\sigma}\downarrow_T$) to denote the pre-state at timestamp $\tau^\dagger$ defined by restricting $\hat{h}$ and $\hat{k}$ to the timestamp $\tau^\dagger$ (resp. the set of polarized timestamps T). This gives a semantics for formulas over $\hat{\Sigma}$: for any $\hat{\sigma} \in \hat{\Sigma}$, and for any A, $\hat{\sigma} \vDash A$ if and only if (1) $logs(\hat{\sigma}) \subseteq \{\mathbf{now}\}$ and (2) $\hat{\sigma}\downarrow_{\mathbf{now}} \vDash A$.

We now extend the logic to be able to talk about logged cells. Note that, within a memory state, the same location may be allocated with a different content for different timestamps, which we call conflicting cells. For $\hat{\sigma} = (s, \hat{h}, \hat{k})$ and a polarized timestamp $\tau^\dagger$, we write $\langle\tau^\dagger\rangle\hat{\sigma}$ for the set of states that result from $\hat{\sigma}$ by tagging all cells with timestamp $\tau^\dagger$, for any possible resolution of conflicting cells. Formally, $\hat{\sigma}' = (s, \hat{h}', \hat{k}') \in \langle\tau^\dagger\rangle\hat{\sigma}$ if $logs(\hat{\sigma}') = \{\tau^\dagger\}$ and for all $l \in Loc$ (resp. for all $\varepsilon \in Endpoint$) there is a timestamp $\tau'^\ddagger$ such that $\hat{h}'(l, \tau^\dagger) = \hat{h}(l, \tau'^\ddagger)$ (resp. $\hat{k}'(\varepsilon, \tau^\dagger) = \hat{k}(\varepsilon, \tau'^\ddagger)$). When there are no conflicting cells, that is when $\langle\tau^\dagger\rangle\hat{\sigma} = \{\hat{\sigma}_0\}$, we write $[\tau^\dagger]\hat{\sigma}$ to denote $\hat{\sigma}_0$. Finally, we write $\langle\tau^\dagger\rangle A$ to denote $\bigsqcup\{\langle\tau^\dagger\rangle\hat{\sigma} \mid \hat{\sigma} \vDash A\}$, and $\langle\tau_1^\dagger * \tau_2^\ddagger\rangle A$ to denote the set of states $\{[\tau_1^\dagger]\hat{\sigma} \bullet [\tau_2^\ddagger]\hat{\sigma} \mid \hat{\sigma} \vDash A\}$.

Finally, we restrict h.p. memory states to well-formed ones in the same way as for memory states, and limit the composition of states so that the logged content of a message is never split into two, nor extended by the frame rule, thus preventing two distinct messages from being logged at the same timestamp.

Definition 2 (H.P. memory states). *The separation subalgebra $(\hat{\Sigma}^{\mathsf{wf}}, \bullet, \hat{u})$ of well-formed h.p. memory states is the subalgebra of $(\hat{\Sigma}, \circ, \hat{u})$ obtained by restricting $\hat{\Sigma}$ to states $\hat{\sigma}$ such that $\mathrm{now}(\hat{\sigma}) \in \Sigma^{\mathsf{wf}}$, and strengthening the compatibility relation $\perp$ by:*
***AtomicLogs**: $\hat{\sigma} \bullet \hat{\sigma}'$ is defined if $\hat{\sigma} \circ \hat{\sigma}' \in \hat{\Sigma}^{\mathsf{wf}}$ and for all $\dagger \in \{?, !\}$ and $\tau \neq \mathbf{now}$, $[\mathbf{now}](\hat{\sigma}\downarrow_{\tau^\dagger}) \vDash \mathsf{emp}$ or $[\mathbf{now}](\hat{\sigma}'\downarrow_{\tau^\dagger}) \vDash \mathsf{emp}$.*

Lemma 2. *$(\hat{\Sigma}^{\mathsf{wf}}, \bullet, \hat{u})$ is a separation algebra.*

Finally, we extend the satisfaction relation of Sec. 2.3 to h.p. states by overloading every predicate but $\overset{ep}{\mapsto}$ and every constructor in the obvious way. We overload $\overset{ep}{\mapsto}$ with two new meanings:

$$(s, \hat{h}, \hat{k}) \vDash E \overset{ep}{\mapsto} (C\{a\}, E', n_?, n_!, \ell_?, \ell_!) \text{ iff}$$
$$\begin{cases} v(E, E') \subseteq dom(s) \;\&\; dom(\hat{k}) = \{([\![E]\!]s, \mathbf{now})\} \;\&\; dom(\hat{h}) = \emptyset \\ \;\&\; \hat{k}([\![E]\!]s, \mathbf{now}) = (C, a, [\![E']\!]s, n_?, n_!, \ell_?, \ell_!) \end{cases}$$
$$(s, \hat{h}, \hat{k}) \vDash E \overset{ep}{\mapsto} (C\{a\}, E') \text{ iff } \exists n_?, n_!, \ell_?, \ell_!.\ (s, \hat{h}, \hat{k}) \vDash E \overset{ep}{\mapsto} (C\{a\}, E', n_?, n_!, \ell_?, \ell_!)$$

4.2 Refined small axioms

We define the semantics $(\!|c|\!) : \hat{\Sigma}^{\mathsf{wf}} \to \mathcal{P}(\hat{\Sigma}^{\mathsf{wf}})^\top$ of an atomic command c as the best local action of a specification $\hat{\phi}$ over $\hat{\Sigma}^{\mathsf{wf}}$. For most of the commands, $\hat{\phi}$ is simply the specification ϕ given by the small axiom associated to it in the proof system, interpreted over $\hat{\Sigma}^{\mathsf{wf}}$ (according to the already mentioned interpretation of $\hat{\sigma} \vDash A$: $logs(\hat{\sigma}) = \{\mathbf{now}\}$ and $\mathrm{now}(\hat{\sigma}) \vDash A$). The only commands for which $\hat{\phi} \neq \phi$ are `open`, `send` and `receive`, which need to deal with histories.

Figure 2 Small axioms of the sub-atomic operations

$$\{O \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon, n_?, n_!, \ell_?, \ell_!) \wedge E' = v\}$$
enq(E,E')
$$\{O \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon, n_?, n_! + 1, \ell_?, \ell_!) \wedge E' = v \wedge \ell_![n_!] = (-, v, -)\}$$

$$\{O, x \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon, n_?, n_!, \ell_?, \ell_!)\}$$
x = deq(E)
$$\{O, x \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon, n_? + 1, n_!, \ell_?, \ell_!) \wedge \ell_?[n_?] = (-, x, -)\}$$

$$\frac{a \overset{\dagger m}{\longrightarrow} b \in C}{\{O \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon, n_?, n_!, \ell_?, \ell_!)\}}$$
contract †(m,E)
$$\{O \Vdash E \overset{ep}{\mapsto} (C\{b\}, \varepsilon, n_?, n_!, \ell_?, \ell_!) \wedge \ell_\dagger[n_\dagger] = (m, -, -)\}$$

$$\{O, x \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon, n_?, n_!, \ell_?, \ell_!)\}$$
x = cur_ts †(E)
$$\{O, x \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon, n_?, n_!, \ell_?, \ell_!) \wedge \ell_\dagger[n_\dagger - 1] = (-, -, x)\}$$

$$\{O, e \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon, n_?, n_!, \ell_?, \ell_!)\}\ \mathsf{e} = \text{peer(E)}\ \{O, e \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon, n_?, n_!, \ell_?, \ell_!) \wedge e = \varepsilon\}$$

$$\{O \Vdash E = \varepsilon \wedge E' = v \wedge \mathsf{emp}\}\ \text{new(m,E,E')}\ \{O \Vdash E = \varepsilon \wedge E' = v \wedge I_m(\varepsilon, v)\}$$

$$\{O \Vdash I_m(E, E')\}\ \text{free (m,E,E')}\ \{O \Vdash \mathsf{emp}\} \qquad \frac{\hat{\sigma} \vDash O \Vdash I_m(E, E') \wedge t = \tau}{\{\hat{\sigma}\}\ \log^\dagger\text{(m,E,E',t)}\ \{\langle \mathbf{now} * \tau^\dagger \rangle \hat{\sigma}\}}$$

The semantics of `open` is the simplest one. As mentioned in Sec. 3.4, the histories attached to the endpoints are guessed when they are created, and dual histories should match; moreover, the queues' counters are initialized to zero. The refined small axiom for `open` is hence the following:

$$\frac{i = \mathsf{init}(C)}{\{e, f \Vdash \mathsf{emp}\}\ (\mathsf{e},\mathsf{f})\ = \text{open(C)}\ \{e, f \Vdash e \overset{ep}{\mapsto} (C\{i\}, f, 0, 0, \ell, \ell') * f \overset{ep}{\mapsto} (\bar{C}\{i\}, e, 0, 0, \ell', \ell)\}}$$

The semantics of `send` and `receive` are more complex: they are the composition of several sub-atomic operations that perform basic tasks.

```
send(m, E, E') ≜ atomic { contract!(m,E); enq(E,E'); local t in {
                          t = cur_ts!(E); log!(m,E,E',t); free(m,E,E');}}
x = receive(m,E) ≜ atomic {contract?(m,E); x = deq(E);
                           local t,e in { t = cur_ts?(E); e = peer(E);
                            new(m,e,x); log?(m,e,x,t);}}
```

Intuitively, `contract`† checks whether the contract authorizes the communication, `enq` and `deq` are the pure pointer passing counterparts of `send` and `receive`, `cur_ts`† selects in the history which timestamp to use for logging the current communication, `peer(E)` retrieves the peer of E, `log`† logs a copy of the part of the heap that is transferred, and `new` and `free` allocate and deallocate this transferred heap. Fig 2 presents the small axioms defining these sub-atomic operations.

4.3 Soundness

In order to establish the soundness for the whole proof system, all we have to do is to establish the soundness of all atomic commands with respect to their coarse small

axioms. We say that a local function f over $\hat{\Sigma}^{\mathsf{wf}}$ satisfies a specification ϕ over Σ^{wf} and write $f \vDash \phi$ if for all $(A, B) \in \phi$, $\langle\langle A \rangle\rangle\ f\ \langle\langle B \rangle\rangle$. Let $\mathrm{now}^{-1}(A)$ denote the set of all $\hat{\sigma} \in \hat{\Sigma}^{\mathsf{wf}}$ such that $\hat{\sigma} = \hat{\sigma}\downarrow_{\mathbf{now}}$ and $\mathrm{now}(\hat{\sigma}) \in A$.

Definition 3 (Implementation). *A specification $\hat{\phi}$ over $\hat{\Sigma}^{\mathsf{wf}}$ implements a specification ϕ over Σ^{wf} if* $\forall (A, B) \in \phi.\ \exists (\hat{A}, \hat{B}) \in \hat{\phi}.\ \mathrm{now}^{-1}(A) \sqsubseteq \hat{A}\ \&\ \mathrm{now}(\hat{B}) \sqsubseteq B$.

Lemma 3. *If $\hat{\phi}$ implements ϕ, then for all local function f, $f \vDash \hat{\phi}$ implies $f \vDash \phi$.*

We can show that the refined small axiom of `open` implements the corresponding coarse axiom, and that sub-atomic commands implement some specifications which, composed together, allow us to derive the coarse small axioms of `send` and `receive`. Abstract separation logic allows us to conclude that our proof system is sound.

Lemma 4 (Soundness for atomic commands). *If $\{A\}\ c\ \{B\}$ is an axiom of our proof system then for all $\hat{\sigma}$ such that* $\mathrm{now}(\hat{\sigma}) \vDash A$, $\mathrm{now}(\llparenthesis c \rrparenthesis(\hat{\sigma})) \sqsubseteq B$.

Theorem 1 (Soundness). $\vdash \{A\}\ p\ \{B\}$ *implies* $\vDash \{A\}\ p\ \{B\}$.

We can easily derive from this theorem that in every provable program, there is no memory violation or race, and contracts are respected. Memory leaks are not yet guaranteed to be avoided: this is the purpose of the transfers erasure property.

5 Transfers erasure property

In this section, we relate the transferring, local semantics we introduced for establishing the soundness of our proof system to the intended non-transferring, global semantics. Defining the non-transferring semantics is rather simple thanks to our decomposition of `send` and `receive` in sub-atomic operations. `check_inv` is added so that $\texttt{send}_g^{nt}$ still faults whenever the invariant of the message is not satisfied.

Definition 4 (Non-transferring semantics). *The non-transferring semantics $\llparenthesis . \rrparenthesis^{nt}$ is the semantics that differs from $\llparenthesis . \rrparenthesis$ by erasing transfers in* `send` *and* `receive`*:*

$$\texttt{send(m,E,E')}^{nt} \triangleq \texttt{atomic}\ \{\ \texttt{contract}^{!}\texttt{(m,E);enq(E,E');check\_inv(m,E,E');}\ \}$$

$$\texttt{x=receive(m,E)}^{nt} \triangleq \texttt{atomic}\ \{\texttt{contract}^{?}\texttt{(m,E); x = deq(E);}\ \}$$

$$c^{nt} \triangleq c \quad \textit{otherwise}$$

where `check_inv(m,E,E')` *is the best local action defined by the Hoare triples* $\{\hat{\sigma}\}$ *check_inv (m,E,E')* $\{\hat{\sigma}\}$ *for all $\hat{\sigma}$ satisfying $I_m(E, E')$.*

In order to relate $\llparenthesis . \rrparenthesis$ and $\llparenthesis . \rrparenthesis^{nt}$, we first need to restrict them to well-interleaved local traces, otherwise many undesired executions would have to be considered: a receive may precede a send, or the message that is sent may not necessarily be the same as the one that is received. This can be observed directly on the resulting memory states thanks to histories, so restricting the semantics to legal states is enough to rule out executions that do not comply with the intended global semantics.

Let $UL(\hat{\sigma})$ denote the set of unmatched logs of $\hat{\sigma}$, that is $UL(\hat{\sigma}) = \{\tau^{\dagger} \in \mathit{logs}(\hat{\sigma}) \mid \tau^{\bar{\dagger}} \notin \mathit{logs}(\hat{\sigma})\}$, and let $\mathit{transfer}(\hat{\sigma})$ denote $\hat{\sigma}\downarrow_{UL(\hat{\sigma})}$. A state $\hat{\sigma}$ is *partitioned* if and only

if $\langle \mathbf{now} \rangle transfer(\hat{\sigma}) = \{\hat{\sigma}_0\}$ and $\hat{\sigma} \lfloor_{\mathbf{now}} \perp \hat{\sigma}_0$. When this is the case, the closure of $\hat{\sigma}$ is defined as $closure(\hat{\sigma}) \triangleq \hat{\sigma} \lfloor_{\mathbf{now}} \bullet[\mathbf{now}] transfer(\hat{\sigma})$. A legal state should always be partitioned (intuitively, a cell cannot be both in transfer and owned by a thread), the logged contents of dual messages should match, and the read history of any endpoint should have been played at most up to the same point as the write history of its peer. Moreover, all timestamps should be different, except dual timestamps.

Definition 5 (Legal state). *A state* $\hat{\sigma} = (s, \hat{h}, \hat{k})$ *is* legal *when it satisfies*

Partitions $\hat{\sigma} \in \hat{\Sigma}^{\mathsf{wf}}$ *is partitioned*

DualMatch $\forall \tau.\, [\mathbf{now}](\hat{\sigma} \lfloor_{\tau^?}) \vDash \neg \mathsf{emp} \Rightarrow \hat{\sigma} \lfloor_{\tau^!} = \hat{\sigma} \lfloor_{\tau^?}$

Asynch $closure(\hat{\sigma}) \vDash \begin{pmatrix} \varepsilon \overset{ep}{\mapsto} (-, \varepsilon', n_?, -, -, -) * \\ \varepsilon' \overset{ep}{\mapsto} (-, \varepsilon, -, n_!, -, -) * \mathsf{true} \end{pmatrix} \Rightarrow n_! \geq n_?$

DisjointLogs $\forall \varepsilon, \varepsilon', \varepsilon''.\, \begin{cases} k(\varepsilon) = (-,-,-,-,\ell_?,\ell_!) \;\&\; k(\varepsilon') = (-,\varepsilon'',-,-,\ell'_?,\ell'_!) \\ \&\; \varepsilon \neq \varepsilon' \;\&\; \varepsilon \neq \varepsilon'' \end{cases}$
$\Rightarrow$ *all timestamps appearing in* $ts(\ell_?),\ ts(\ell_!),\ ts(\ell'_?),\ ts(\ell'_!)$ *are distinct*

The global semantics is then defined as $(\!|c|\!)_g(\hat{\sigma}) \triangleq \{\hat{\sigma}' \in (\!|c|\!)(\hat{\sigma}) \mid \hat{\sigma}' \text{ legal}\}$ if $(\!|c|\!)(\hat{\sigma}) \neq \top$, $\top$ otherwise. The global non-transferring semantics $(\!|.|\!)^{nt}_g$ is defined the same way. Our aim is to show that the global semantics is the same as the non-transferring one, up to a closure that brings back cells that are being transferred. It might be a surprise that this result does not hold without some particular restrictions on the contracts that ensure that no messages are lost when a channel is closed. We do not give the most general condition on contracts that achieves this non-leaking property, but rather provide a sufficient condition that is easy to check syntactically on the contract. These restrictions are very similar to those used in Singularity. A contract is *deterministic* if any two distinct edges with the same source have different labels. It is *positional* if every two edges with the same source are labeled with either two sends or two receives. A state is *synchronizing* if every graph cycle that goes through it contains at least one send and one receive.

Definition 6. *A contract is* non-leaking *if it is deterministic, positional, and every final state is a synchronizing state.*

For instance, the contract of the example is non-leaking, but would be leaking if all states were merged in a single state.

Theorem 2. *For any provable program* p *with non-leaking contracts, for all* $tr \in T(p)$, $[\![tr]\!]^{nt}_g(u) = closure([\![tr]\!]_g(u))$.

Remark 1. In particular, if $\vdash \{\mathsf{emp}\}\ p\ \{\mathsf{emp}\}$ and p terminates, then p does not fault on memory accesses nor leaks memory for any of the considered semantics.

We establish this result by induction on tr with a stronger inductive property. Due to lack of space, we do not detail the rather involved proof. One hard part of the proof, as mentioned earlier, is to establish that no memory is leaked when a channel is closed. Since non-leaking contracts are deterministic and positional, it can be proved that channels are in fact half-duplex. Moreover, as contracts are respected, in any reachable

memory state, and for any coupled endpoints $\varepsilon, \varepsilon'$ of this state, the list of unread messages by ε, if not empty, is the same as the one labeling a read path from the contract's state of ε to the one of ε'. We then prove the absence of memory leak by the following argument: as a channel is closed if and only if the two endpoints are in the same final state, their histories may differ only from a read or a write cycle in the contract, and since final states are synchronizing, this cycle must be the empty cycle.

Conclusion and future work

We presented a proof system for copyless message passing ruled by contracts, illustrating how contracts may facilitate the work of the `Sing#` compiler in the static analysis that verifies the absence of ownership violations. We established the soundness of the proof system with respect to an over-approximating local semantics where message exchanges are unsynchronized, and restricted it to a global semantics for which we established the transfers erasure property.

We illustrated our proof system on a small and rather simple example. We focused on the foundations of our proof system in this work, but we wish to tackle more case-studies in the future. We moreover plan to automate the proof inference using an existing tools like `Smallfoot` [1]. Another challenging application of our proof system could be to prove a distributed garbage collector synchronized by message passing, for which the transfers erasure property would potentially be an important issue.

References

1. `http://www.dcs.qmul.ac.uk/research/logic/theory/projects/smallfoot/`.
2. R. Bornat, C. Calcagno, and H. Yang. Variables as Resource in Separation Logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006.
3. S. Brookes. A semantics for concurrent separation logic. *TCS*, 375(1-3):227–270, 2007.
4. C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular Safety Checking for Fine-Grained Concurrency. *Lecture Notes in Computer Science*, 4634:233, 2007.
5. Cristiano Calcagno, Peter O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *22nd LICS*, pages 366–378, 2007.
6. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, 2006.
7. Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, LNCS, pages 19–37, 2007.
8. Tony Hoare and Peter O'Hearn. Separation logic semantics for communicating processes. *Electron. Notes Theor. Comput. Sci.*, 212:3–25, 2008.
9. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In Jan Vitek, editor, *ECOOP*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
10. P.W. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 375(1-3):271–307, 2007.
11. D. Pym and C. Tofts. A Calculus and logic of resources and processes. *Formal Aspects of Computing*, 18(4):495–517, 2006.
12. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*.
13. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-Based Language and Its Typing System. *Lecture Notes in Computer Science*, pages 398–398, 1994.