Proving Copyless Message Passing

Étienne Lozes[1] Jules Villard[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. Moreover, forcing programmers to describe the underlying protocol governing communications may help detect deadlocks at a higher level.

The Singularity operating system [6] is a prominent application of these ideas. It can safely run processes sharing a unique address space without memory protection. Executable processes are written in the `Sing#` programming language, which supports (copyless) message passing primitives. Ownership violations are detected at compile-time using static analysis techniques, which is essential for preserving system's integrity. Moreover, communications are ruled by contracts, a form of session types supported by `Sing#`, and this feature seems to be essential for the static analysis. To our knowledge, no formal presentation of how the contracts interact with the static analysis has ever been presented, nor a semantics for copyless message passing primitives.

The goal of this paper is thus to give a semantics for these programs, as well as a proof system that validates ownership transfers and contract obedience, giving a higher view of the work that has to be achieved by this static analysis. Our idealized programming language allows memory manipulation and asynchronous communications ruled by contracts, following the ideas of `Sing#`. However, we chose to be able to detect memory leaks, whereas `Sing#` is equipped with a garbage collector, and we support complete mobility of channels, similar to π-calculus, where `Sing#` provides internal mobility only.

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, they have a non-trivial impact on the absence of memory leaks when channels are closed.

Our first contribution is the proof of soundness for our proof system. Our soundness result implies that provable programs do not fault on memory accesses, are race free, and contract obedient. Unlike for concurrent separation logic, it cannot entail the absence of memory leaks, due to the complexity of the shared resources in our setting.

Our second contribution is to propose a new approach for defining a sound semantics for any proof system based on separation logic that deals with ownership transfers. 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, for which the absence of memory leaks during transfers is possible to state. However, neither the local nor the global semantics reflect the intended semantics.

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, we christened transfers erasure property, relates the global semantics to the intended non-transferring semantics in a non-trivial way. To our knowledge, this is the first time a precise statement and a detailed proof of this result is presented.

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 Section 2, and demonstrate how to prove the example. Section 3 gives an overview of the main ingredients of our semantics. We develop the semantics in more details in Section 4, leading to a soundness result for our logic. Section 5 is devoted to the transfers erasure property.

Related work 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 involve 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.

Session types [13] look closely related to the way contracts are defined. The main differences are that session types are usually devoted to synchronous communications. Since our communications are asynchronous and we want to support local reasoning, our primitives for channel creation and disposal differ from the ones used in the models based on process algebras. Session types were also used in real programming languages, as SessionJ [9], but do not address the problem of copyless message passing.

While our programming language is strongly based on the ideas of the `Sing#` language, there are some differences worth noting between the two of them. As our language is not full-fledged, we did not provide mechanisms for error handling, especially when one endpoint is abruptly closed. On the other hand, we are more permissive about transfers of endpoints, as Sing# only allows to transfer endpoints on which no communication has occurred yet.

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: 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 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|-p->(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.

1.2 Sending a list cell by cell

The imperative programming language we use features standard variable and memory manipulation. We moreover use send(e.m, x) to send message m with value x over endpoint e and x := receive(f.m) 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 enqueing and dequeing over one of the two queues that are shared by two coupled endpoints (one queue for each direction). To fix ideas, we may imagine that every queue of the pair of endpoints (e,f) comes equipped with a semaphore sem, a buffer buf and two global variables i,j, and enq(e.m,v) and x := deq(f.m) could be implemented by

```
#define enq(e.m.v) {buf[i++] := v; up(sem)}
#define x := deq(f.m) {down(sem); x := buf[j++]}
```

Endpoints are allocated on the heap upon channel creation (`(e,f):=open(C)`) and closed together (`close(e,f)`). This differs from session types and the programming languages it inspired, where these operations result from an agreement between two threads over an existing channel. Closing a channel in such a way in an asynchronous setting would lead to the first closing thread to implicitly send a special message to its partner; we chose instead to close both endpoints simultaneously.

Let us now give a program implementing contract `C` by sending a list cell by cell over a channel. The serving endpoint e is held by the `putter` program, which communicates with `getter`. The program is given a list starting at the address pointed to by x that it sends cell by cell over endpoint 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 `//`) are elements of the proof and will be explained later.

```
10 putter(e,x) [e,x||- e|-p->(C[transfer],X) * ls(x)] {
11   local t;
12   while x != null {
13     // e,x,t||- x|->Y * ls(Y) * e|-p->(C[transfer],X)
14     t := *x;
15     send(e.cell, x);
16     // e,x,t||- ls(t) * e|-p->(C[wait_ack],X)
17     x := t;
18     receive(e.ack); }
19   // e,x,t||- e|-p->(C[transfer],X)
20   send(e.close_me, e); } [e,x||- emp]
21
22 getter(f) [f||- f|-p->(~C[transfer],Y)] {
23   local x, e := null;
24   // 0 = x,e,f
25   while e = null {
26     // O||- f|-p->(~C[transfer],Y) * e=0
27     switch receive {
28     x := receive(f.cell): {
29       // O||- f|-p->(~C[wait_ack],Y) * e=0 * x |-> -
30       free(x)
31       // O||- f|-p->(~C[wait_ack],Y) * e=0
32       send(f.ack); }
33     e := receive(f.close_me): {} }}
34   // O||- e|-p->(C[end],f) * f|-p->(~C[end],e)
35   close(e, f); } [f||-emp]
36
37 main() [x||- ls(x)] {
38   local e,f;
39   (e,f) := open(C);
40   // x,e,f||- ls(x) * e|-p->(C[transfer],f)
41   //                * f|-p->(~C[transfer],e)
42   putter(e,x); || getter(f); } [x||- emp]
```

Operationally, `send` and `receive` are not different from enq and deq, but logically they are. In particular, we should not accept as a valid program the one in which lines 14-15 are swapped, because sending is intended to induce the lost of ownership of the cell at address x.

2 A separation logic for copyless message passing

2.1 Syntax of programs

We assume infinite sets $\mathit{Var} = \{e, f, x, y, \dots\}$, $\mathit{Loc} = \{l, \dots\}$, $\mathit{Endpoint} = \{\varepsilon, \dots\}$, $\mathit{MsgId} = \{m, \dots\}$, $\mathit{State} = \{a, b, \dots\}$ and $\mathit{Val} = \{v, \dots\}$ of respectively variables, memory locations, endpoints, message identifiers, contracts' states and values. All sets but values are pairwise disjoint, and $\mathit{Loc} \uplus \mathit{Endpoint} \uplus \{0\} \subseteq \mathit{Val}$. The grammar of expressions, boolean expressions, atomic commands and programs is as follows:

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

$$\begin{aligned} c ::= {}& \texttt{assume(B)} \mid \texttt{x := E} \mid \texttt{x := new()} \mid \texttt{*E := E} \mid \texttt{x := *E} \mid \texttt{free(E)} \\ & \mid \texttt{(e,f) := open(C)} \mid \texttt{close(E,E)} \mid \texttt{send(E.m, E)} \mid \texttt{x := receive(E.m)} \end{aligned}$$

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

`assume(B)` blocks unless B holds. 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(E.m);` p`}` for every `x := receive(E.m):` p of its body. We leave the similar definitions of `while` loops and `if` statements to the attention of the reader. We have split the example program of Sec. 1.2 into subroutines `putter` and `getter`, but one should actually inline these procedures 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 $\mathit{LVar} = \{X, Y, \dots\}$ 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:

$$\begin{aligned} A ::= {}& \mathsf{emp_s} \mid \mathsf{own}(x) \mid E = E && \text{stack predicates} \\ & \mid \mathsf{emp_h} \mid \mathsf{emp_{ep}} \mid E \mapsto E \mid E \overset{ep}{\mapsto} (C[a], E) && \text{heaps predicates} \\ & \mid \neg A \mid A \wedge A \mid \exists X.\, A \mid A * A \mid A \mathrel{-\!\!*} A && \text{connectives} \end{aligned}$$

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

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

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.

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

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 $I_m[\texttt{src} \leftarrow E_1, \texttt{val} \leftarrow E_2]$ as $I_m(E_1, E_2)$. Figure 1 presents all the

Figure 1 Proof System Rules

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

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

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

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

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

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

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

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

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

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

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(E.m,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 $\{O \Vdash E \overset{ep}{\mapsto} (C[a], \varepsilon) * (E \overset{ep}{\mapsto} (C[b], \varepsilon) \mathbin{-\!\!*} I_m(E, F))\}$ send(E.m,F) $\{O \Vdash \mathsf{emp}\}$ (taking $A = \mathsf{emp}$). We will write $\vdash \{A\}$ p $\{B\}$ when this triple is derivable.

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 $\{x \Vdash \mathrm{ls}(x)\}\ p\ \{x \Vdash \mathsf{emp}\}$. Bracketed formulas are used to denote the pre and post-condition of a program. We start with the precondition $x \Vdash \mathrm{ls}(x)$ of the program, where $\mathrm{ls}(x)$ is the inductive list predicate verifying emp if $x = 0$ and $\exists X. x \mapsto X * \mathrm{ls}(X)$ otherwise. Before entering the parallel composition, we obtain $x, e, f \Vdash \mathrm{ls}(x) * e \overset{ep}{\mapsto} (C[\mathtt{transfer}], f) * f \overset{ep}{\mapsto} (\bar{C}[\mathtt{transfer}], e)$. To apply the rule for parallel composition, we have to split the state into two. `putter` will get resources $e, x \Vdash e \overset{ep}{\mapsto} (C[\mathtt{transfer}], -) * \mathrm{ls}(x)$ and `getter` $f \Vdash f \overset{ep}{\mapsto} (\bar{C}[\mathtt{transfer}], -)$. The next important step is after the loop, at line 18; 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 $\sigma_1 \perp \sigma_2$ 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 $\sigma \# \sigma'$ 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.

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* $\alpha \in \{c, \text{norace}(c_1, c_2), \mathrm{n}_x, \mathrm{d}_x\}$ 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.

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

$$T(p_1; p_2) = \{\mathit{tr}_1; \mathit{tr}_2 \mid \mathit{tr}_i \in T(p_i)\} \qquad T(p_1 \parallel p_2) = \{\mathit{tr}_1\ \mathit{zip}\ \mathit{tr}_2 \mid \mathit{tr}_i \in T(p_i)\}$$

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

Parallel composition is treated as a syntactic interleaving of commands. We force all racy programs to fault by placing $\text{norace}(c_1, c_2)$ 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 $\varepsilon\ \mathit{zip}\ \mathit{tr} = \mathit{tr}\ \mathit{zip}\ \varepsilon = \mathit{tr}$ in the base case, and by $(c_1; \mathit{tr}_1)\,\mathit{zip}\,(c_2; \mathit{tr}_2) = \text{norace}(c_1, c_2); ((c_1; (\mathit{tr}_1\,\mathit{zip}\,(c_2; \mathit{tr}_2))) \cup (c_2; (c_1; \mathit{tr}_1)\,\mathit{zip}\,\mathit{tr}_2))$.

Semantics The denotational semantics of traces is the composition of the interpretation of atomic actions: $[\![\alpha]\!] = (\!|\alpha|\!)$ and $[\![\mathit{tr}_1; \mathit{tr}_2]\!] = [\![\mathit{tr}_1]\!]; [\![\mathit{tr}_2]\!]$. This assumes that a semantics $(\!|c|\!)$ is defined for all primitive commands, which we will give later. The semantics of $\text{norace}(c_1, c_2)$ 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 $(\!|\mathrm{d}_x|\!) \triangleq d_x \triangleq \mathit{bla}[\mathsf{own}(x), \mathsf{emp_s}]$.

Limitations of the trace semantics Following this approach is essential for deriving easily the soundness of the parallel rule: $[\![p \,\|\, p']\!](\sigma \bullet \sigma') \sqsubseteq [\![p]\!](\sigma) * [\![p']\!](\sigma')$. 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.

Our solution is to first consider an over-approximating semantics $[\![.]\!]$ for which the soundness can already be established, and then restricts it to a global semantics $[\![.]\!]_g$ which will model the intended semantics. To do so, we will define an history-preserving memory model $\hat\Sigma^{\mathsf{wf}}$ such that all communications that occur during tr with initial state $\hat{\sigma_0}$ can be observed in any final state $\hat\sigma \in [\![\mathit{tr}]\!](\hat{\sigma_0})$. We will also define a notion of *legal memory state* $\hat\sigma$ for which the history of communications must be coherent, and then define the global semantics of commands (leaving the trace semantics unchanged) as:

$$(\!|c|\!)_g(\hat{\sigma}) \triangleq \begin{cases} \{\hat{\sigma}' \in (\!|c|\!)(\hat{\sigma}) \mid \hat{\sigma}' \text{ legal}\} & \text{if } (\!|c|\!)(\hat{\sigma}) \neq \top \\ \top & \text{otherwise} \end{cases}$$

We will explain now what information histories will have to contain in the h.p. model. Legal states will be defined 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 eliminating ill-synchronized traces, *i.e.* traces where two critical sections over the same resource are 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 by making them block when executed ($\{A\}$ p $\{B\}$ will be true when p does not terminate nor fault). 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 are thus 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 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 $[\tau^\dagger]\hat{\sigma}$, 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 endpoint 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 will have to be logged. The peer 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. Histories of a channel will be guessed upon opening of the channel and will remain unchanged until the channel will be closed.

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 $\mathit{Hist} \triangleq (\mathit{MsgId} \times \mathit{Val} \times \mathcal{T})^\omega$:

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

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 $\mathit{C\hat{H}eap}$, $\mathit{E\hat{H}eap}$.

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 2 (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 $\mathit{logs}(\hat{\sigma})$ to denote the set of polarized timestamps that appear in $\hat{\sigma}$, *i.e.* $\mathrm{snd}(\mathit{dom}(\hat{h})) \cup \mathrm{snd}(\mathit{dom}(\hat{k}))$.

If $\hat{\sigma} = (s, \hat{h}, \hat{k})$, we write $\hat{\sigma}\lfloor_{\tau^\dagger}$ (resp. $\hat{\sigma}\lfloor_T$) 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) $\hat{\sigma}\lfloor_{\mathbf{now}} \vDash A$.

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 frame rule, thus preventing two distinct messages to be logged at the same timestamp.

Definition 1 (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* $\tau \neq \mathbf{now}$*,* $[\mathbf{now}](\hat{\sigma}\lfloor_{\tau^{\dagger}}) \vDash \mathsf{emp}$ *or* $[\mathbf{now}](\hat{\sigma}'\lfloor_{\tau^{\dagger}}) \vDash \mathsf{emp}$.

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.

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:

Figure 2 Small axioms of the sub-atomic operations

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

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

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

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

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

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

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

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

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

```
      send(E.m,E') ≜ atomic { contract!(E.m); enq(E,E'); local t in {
                       t := cur_ts!(E); log!(m,E,E',t); free(m,E,E');}}
x := receive(E.m) ≜ atomic {contract?(E.m); 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 pure pointer passing as presented in Sec. 1.2, `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 $\text{now}^{-1}(A)$ denote the set of all $\hat\sigma \in \hat\Sigma^{\mathsf{wf}}$ such that $\hat\sigma = \hat\sigma\lfloor_{\mathbf{now}}$ and $\text{now}(\hat\sigma) \in A$.

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

Lemma 4 *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`.

Lemma 5 (Soundness for atomic commands) *If $\{A\}\ c\ \{B\}$ is an axiom of our proof system then for all $\hat{\sigma}$ such that* $\operatorname{now}(\hat{\sigma}) \vDash A$, $\operatorname{now}((\!|c|\!)(\hat{\sigma})) \sqsubseteq B$.

Abstract separation logic allows us to conclude that our proof system is sound.

Theorem 6 (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 semantics we introduced for establishing the soundness of our proof system to the intended non-transferring 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 `send`$_g^{nt}$ still faults whenever the invariant of the message is not satisfied.

Definition 5.1 (Non-transferring semantics). *The non-transferring semantics $(\!|.|\!)^{nt}$ is the semantics that differs from $(\!|.|\!)$ by erasing transfers in* `send` *and* `receive`*:*

`send(E.m,E')`$^{nt} \triangleq$ `atomic` $\{$ `contract`$^{!}$`(E.m);enq(E,E');check_inv(m,E,E');` $\}$
`x:=receive(E.m)`$^{nt} \triangleq$ `atomic` $\{$`contract`$^{?}$`(E.m); x := deq(E);` $\}$
$c^{nt} \triangleq c$ *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 $(\!|.|\!)$ and $(\!|.|\!)^{nt}$, 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 $UL(\hat{\sigma}) = \{\tau^{\dagger} \in \mathit{logs}(\hat{\sigma}) \mid \tau^{\overline{\dagger}} \notin \mathit{logs}(\hat{\sigma})\}$, and let $\mathit{transfer}(\hat{\sigma})$ denote $\hat{\sigma}\downharpoonleft_{UL(\hat{\sigma})}$. A state $\hat{\sigma}$ is *partitioned* if and only if $\langle \mathbf{now} \rangle \mathit{transfer}(\hat{\sigma}) = \{\hat{\sigma}_0\}$ and $\hat{\sigma} \downharpoonleft_{\mathbf{now}} \perp \hat{\sigma}_0$. When this is the case, the closure of $\hat{\sigma}$ is defined as $\mathit{closure}(\hat{\sigma}) \triangleq \hat{\sigma} \downharpoonleft_{\mathbf{now}} \bullet [\mathbf{now}] \mathit{transfer}(\hat{\sigma})$. 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. This is a natural restriction, because in states not complying with it, write transfers may block, thus ruling out the corresponding traces, whereas these traces would be valid in our non-transferring semantics, thus invalidating our correspondence theorem below.

Definition 1 (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 $\forall \tau.\ [\mathbf{now}](\hat{\sigma}\downharpoonright_{\tau^?}) \vDash \neg\mathsf{emp} \Rightarrow \hat{\sigma}\downharpoonright_{\tau^!} = \hat{\sigma}\downharpoonright_{\tau^?}$

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

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 $(\!|.|\!)_g^{nt}$ 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. 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 2 *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 3 *For any provable program* p *with non-leaking contracts, for all* $tr \in T(p)$, $[\![tr]\!]_g^{nt}(u) = closure([\![tr]\!]_g(u))$.

Remark 5.1. In particular, if $\vdash \{\mathsf{emp}\}\ \mathtt{p}\ \{\mathsf{emp}\}$, then p does not fault on memory accesses nor leaks memory if it terminates, 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 study some code of Singularity OS in the future. We moreover plan to automate the proof inference relying on existing tools like `Smallfoot` [1]. Another challenging application of our proof system could be to prove a parallel distributed garbage collectors 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.

A Proof of Soundness

Lemma 4. *If $\hat{\phi}$ implements ϕ, then for all local function f, $f \vDash \hat{\phi}$ implies $f \vDash \phi$.*

Proof It is enough to prove that $bla[\hat{\phi}] \vDash \phi$. Let (A_0, B_0) be some axiom of ϕ. Let $\hat{\sigma} \in \hat{\Sigma}^{\mathsf{wf}}$ be such that $\mathrm{now}(\hat{\sigma}) \vDash A_0$. We have to show that $\mathrm{now}(bla[\hat{\phi}](\hat{\sigma})) \sqsubseteq B_0$.

Let us first assume that $\hat{\sigma} = \hat{\sigma}\lfloor_{\mathbf{now}}$. By definition of implementation, there is at least one $(\hat{A}_0, \hat{B}_0) \in \hat{\phi}$ such that $\hat{\sigma} \in \hat{A}_0$ and $\mathrm{now}(\hat{B}_0) \sqsubseteq B_0$. Then

$$\begin{aligned} bla[\hat{\phi}](\hat{\sigma}) &= \bigsqcap_{(\hat{A},\hat{B})\in\hat{\phi}} \; \bigsqcap_{\hat{\sigma}'\preceq\hat{\sigma},\hat{\sigma}'\in\hat{A}} \hat{B} * \{\hat{\sigma} - \hat{\sigma}'\} \\ &\sqsubseteq \bigsqcap_{\hat{\sigma}'\preceq\hat{\sigma},\hat{\sigma}'\in\hat{A}_0} \hat{B}_0 * \{\hat{\sigma} - \hat{\sigma}'\} \\ &\sqsubseteq \hat{B}_0 \end{aligned}$$

and $\mathrm{now}(\hat{B}_0) \sqsubseteq B_0$.

Let us now assume that $\hat{\sigma} = \hat{\sigma}\lfloor_{\mathbf{now}} \bullet \hat{\sigma}'$. Then $\mathrm{now}(\hat{\sigma}\lfloor_{\mathbf{now}}) = \mathrm{now}(\hat{\sigma}) \in A$, and

$$\mathrm{now}(bla[\hat{\phi}](\hat{\sigma})) \sqsubseteq \mathrm{now}\big(bla[\hat{\phi}](\hat{\sigma}\lfloor_{\mathbf{now}}) * \{\hat{\sigma}'\}\big) = \mathrm{now}\big(bla[\hat{\phi}](\hat{\sigma}\lfloor_{\mathbf{now}})\big)$$

which ends the proof. □

We show that the refined versions of `open`, `send` and `receive` satisfy the Hoare triples of the proof system. As all specifications involved are singletons, the definition 3 becomes "$(\hat{A}, \hat{B})$ implements (A, B) iff $\mathrm{now}^{-1}(A) \sqsubseteq \hat{A}$ & $\mathrm{now}(\hat{B}) \sqsubseteq B$."

Lemma 1 `open`*'s refined small axiom implements the corresponding small axiom of the proof system.*

Proof

- They both have $e, f \Vdash \mathsf{emp}$ as precondition;
- If $\hat{\sigma} \vDash e, f \Vdash e \overset{ep}{\mapsto} (C[i], f, 0, 0, \ell, \ell') * f \overset{ep}{\mapsto} (\bar{C}[i], e, 0, 0, \ell', \ell)$, then $\mathrm{now}(\hat{\sigma}) \vDash e, f \Vdash e \overset{ep}{\mapsto} (C[i], f) * f \overset{ep}{\mapsto} (\bar{C}[i], e)$. □

Lemma 2 `contract`†`(E.m)` *implements*

$$\frac{a \xrightarrow{\dagger m} b \in C}{\{O \Vdash E \overset{ep}{\mapsto} (C[a], \varepsilon)\} \; \textit{contract}\,(E.m) \; \{O \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon)\}}$$

Proof They have the same precondition, and if `contract`†`(E.m)` does not block (*i.e.* if the $\dagger$-history of E contains m indeed at $n_\dagger$) then it ends up in a state satisfying the post-condition. □

Lemma 3 `enq(E,E')` *implements*

$$\{O \Vdash E \overset{ep}{\mapsto} (C[a], \varepsilon) \wedge E' = E'\} \; \textit{enq}(E,E') \; \{O \Vdash E \overset{ep}{\mapsto} (C[a], \varepsilon)\}$$

Proof Similar to `contract`†. □

Lemma 4 `x := deq(E)` *implements*

$$\{O, x \Vdash E \overset{ep}{\mapsto}(C[a], \varepsilon)\}\ x\ := \textit{deq(E)}\ \{O, x \Vdash E \overset{ep}{\mapsto}(C[a], \varepsilon)\}$$

Proof Similar to `contract`†. □

Lemma 5 `x := cur_ts(E)` *implements*

$$\{O, x \Vdash E \overset{ep}{\mapsto}(C[a], \varepsilon)\}\ x\ := \textit{cur\_ts (E)}\ \{O, x \Vdash E \overset{ep}{\mapsto}(C[a], \varepsilon)\}$$

Proof Similar to `contract`†. □

Lemma 6 `e := peer(E)` *implements*

$$\{O, e \Vdash E \overset{ep}{\mapsto}(C[a], \varepsilon)\}\ e\ := \textit{peer(E)}\ \{O, e \Vdash E \overset{ep}{\mapsto}(C[a], \varepsilon) \wedge e = \varepsilon\}$$

Proof Trivial. □

Lemma 7 `log`†`(m,E,E',t)` *implements*

$$\{O, t \Vdash I_m(E, E')\}\ \textit{log}^{\dagger}\textit{(m,E,E',t)}\ \{O, t \Vdash I_m(E, E')\}$$

Proof The preconditions match and if $\hat{\sigma} \vDash O, t \Vdash I_m(E, E')$ then $\langle \textbf{now} * \tau^{\dagger} \rangle \hat{\sigma} \vDash O, t \Vdash I_m(E, E')$. □

We now present the proof of Lemma 5.

Proof

- For all commands but `open`, `send` and `receive`, the result holds as the best local action of a specification satisfies this specification.
- For the `open` command, this is a direct consequence of Lemma 1 and Lemma 4.
- For `send` and `receive`, we prove the small axioms are sound by using the sequential rule of separation logic. We then use Lemma 4 to conclude.

For the `send` command, we have to show that

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

We can derive this Hoare triple using the specifications of the sub-atomic commands described in the previous lemmas. The proof has the following structure:

$$\dfrac{(1) \qquad \dfrac{(2) \qquad \dfrac{\dfrac{\dfrac{(3) \qquad (4)}{\{A_3\}\ c_3;c_4;\ \{A_4\}}\ \textsc{Seq}}{\{A_2\}\ \text{local}\ \ \text{ev}\ \ \text{in}\ \ \{c_3;c_4;\}\ \{A_5\}}\ \textsc{Local} \qquad (5)}{\{A_2\}\ \text{local}\ \ \text{ev}\ \ \text{in}\ \ \{c_3;c_4;\}c_5\ \{A_6\}}\ \textsc{Seq}}{\{A_2\}\ c_2;\ \text{local ev}\ \ \text{in}\ \ \{c_3;c_4;\}c_5\ \{A_6\}}\ \textsc{Seq}}{\{A_1\}\ c_1;c_2;\ \text{local ev in}\ \{c_3;c_4;\}c_5\ \{A_6\}}\ \textsc{Seq}$$

We've used the following shorthands:

$$\begin{array}{ll}
A_1 = O \Vdash E \overset{ep}{\mapsto}(C[a],\varepsilon) * (E \overset{ep}{\mapsto}(C[b],\varepsilon) \mathrel{-\!\!*} (I_m(E,F) * A)) & c_1 = \text{contract (E.m)} \\
A_2 = O \Vdash E \overset{ep}{\mapsto}(C[b],\varepsilon) * (E \overset{ep}{\mapsto}(C[b],\varepsilon) \mathrel{-\!\!*} (I_m(E,F) * A)) & c_2 = \text{enq(E,E')} \\
A_3 = \mathsf{own}(ev) * A_2 & c_3 = \text{ev := cur\_ts}^{!}\text{(E)} \\
A_4 = \mathsf{own}(ev) * A_5 & c_4 = \text{ev := log}^{!}\text{(m,E,E',ev)} \\
A_5 = O \Vdash I_m(E,F) * A & c_5 = \text{free (m,E,E')} \\
A_6 = O \Vdash A &
\end{array}$$

Let us now detail the subproofs.

(1)

$$\dfrac{\dfrac{}{\{O \Vdash E \overset{ep}{\mapsto}(C[a],\varepsilon)\}\ \text{contract (E.m)}\ \{O \Vdash E \overset{ep}{\mapsto}(C[b],\varepsilon)\}}\ \text{Lemma 2}}{\begin{array}{c}\{O \Vdash E \overset{ep}{\mapsto}(C[a],\varepsilon) * (E \overset{ep}{\mapsto}(C[b],\varepsilon) \mathrel{-\!\!*} (I_m(E,F) * A))\} \\ \text{contract (E.m)} \\ \{O \Vdash E \overset{ep}{\mapsto}(C[b],\varepsilon) * (E \overset{ep}{\mapsto}(C[b],\varepsilon) \mathrel{-\!\!*} (I_m(E,F) * A))\}\end{array}}\ \textsc{Frame}$$

(2)

$$\dfrac{A_2 \Rightarrow A_2 \wedge F = F \qquad \dfrac{\dfrac{}{\begin{array}{c}\{O \Vdash E \overset{ep}{\mapsto}(C[b],\varepsilon) \wedge F = F\} \\ \text{enq(E,F)} \\ \{O \Vdash E \overset{ep}{\mapsto}(C[b],\varepsilon)\}\end{array}}\ \text{Lemma 3}}{\{A_2 \wedge F = F\}\ \text{enq(E,F)}\ \{A_2\}}\ \textsc{Frame}}{\begin{array}{c}\{O \Vdash E \overset{ep}{\mapsto}(C[b],\varepsilon) * (E \overset{ep}{\mapsto}(C[b],\varepsilon) \mathrel{-\!\!*} (I_m(E,F) * A))\} \\ \text{enq(E,F)} \\ \{O \Vdash E \overset{ep}{\mapsto}(C[b],\varepsilon) * (E \overset{ep}{\mapsto}(C[b],\varepsilon) \mathrel{-\!\!*} (I_m(E,F) * A))\}\end{array}}\ \textsc{Weak}$$

(3)

$$\dfrac{\textsc{Frame}\ \dfrac{\text{Lemma 5}\ \dfrac{}{\begin{array}{c}\{ev, O \Vdash E \overset{ep}{\mapsto}(C[b],\varepsilon)\} \\ \text{ev := cur\_ts (E)} \\ \{ev, O \Vdash E \overset{ep}{\mapsto}(C[b],\varepsilon)\}\end{array}}}{\{A_3\}\ \text{ev := cur\_ts (E)}\ \{A_3\}} \qquad A_3 \Rightarrow A_4}{\{A_3\}\ \text{ev := cur\_ts (E)}\ \{ev, O \Vdash I_m(E,F) * A\}}\ \textsc{Weak}$$

(4)

$$\dfrac{}{\{ev, O \Vdash I_m(E,F) * A\}\ \text{log}^{!}\text{(m,E,F,ev)}\ \{ev, O \Vdash I_m(E,F) * A\}}\ \text{Lemma 7}$$

(5)

$$\dfrac{\dfrac{}{\{O \Vdash I_m(E,F)\}\ \text{free (m,E,F)}\ \{O \Vdash \mathsf{emp}\}}\ \textsc{Ax}}{\{O \Vdash I_m(E,F) * A\}\ \text{free (m,E,F)}\ \{O \Vdash A\}}\ \textsc{Frame}$$

For the `receive` command, we have to show that

$$\{O, x \Vdash E \overset{ep}{\mapsto} (C[a], \varepsilon)\}\ \text{x := } \mathsf{receive}\,(\text{E.m})\ \{O, x \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) * I_m(\varepsilon, x)\}$$

As for `send`, we derive this Hoare triple using the specifications of the sub-atomic commands. The proof is the following:

$$\dfrac{\text{Lem. 2}\ \dfrac{}{\{O, x \Vdash E \overset{ep}{\mapsto} (C[a], \varepsilon)\}\ \mathsf{contract}\,(\text{E.m})\ \{O, x \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon)\}} \qquad (1)}{\{O, x \Vdash E \overset{ep}{\mapsto} (C[a], \varepsilon)\}\ \mathsf{contract}^{?}(\text{E.m});\ \ldots\ \{O, x \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) * I_m(\varepsilon, x)\}}\ \textsc{Seq}$$

(1)

$$\dfrac{\text{Lem. 3}\ \dfrac{}{\{O, x \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon)\}\ \text{x := }\mathsf{deq}(\text{E})\ \{O, x \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon)\}} \qquad (2)}{\{O, x \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon)\}\ \text{x := }\mathsf{deq}(\text{E});\ \ldots\ \{O, x \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) * I_m(\varepsilon, x)\}}\ \textsc{Seq}$$

(2)

$$\dfrac{\dfrac{\textsc{Frame}\ \dfrac{\text{Lem. 5}\ \dfrac{}{\begin{array}{c}\{O, x, t, e \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon)\} \\ \text{t := }\mathsf{cur\_ts}\,(\text{E}) \\ \{O, x, t, e \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon)\}\end{array}}}{} \qquad (3)}{\begin{array}{c}\{O, x, t, e \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) * I_m(E, x)\} \\ \text{t := }\mathsf{cur\_ts}^{?}(\text{E});\ \text{e := }\mathsf{peer}(\text{E});\ \mathsf{new}(\text{m,e,x});\ \mathsf{log}^{?}(\text{m,E,E',t}); \\ \{O, x, t, e \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) * I_m(\varepsilon, x)\}\end{array}}\ \textsc{Seq}}{\begin{array}{c}\{O, x \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) * I_m(E, x)\} \\ \mathsf{local}\ \text{t,e}\ \mathsf{in}\ \{\ \text{t := }\mathsf{cur\_ts}^{?}(\text{E});\ \text{e := }\mathsf{peer}(\text{E});\ \mathsf{new}(\text{m,e,x});\ \mathsf{log}^{?}(\text{m,e,x,t});\} \\ \{O, x \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) * I_m(\varepsilon, x)\}\end{array}}\ \textsc{Local}$$

(3)

$$\dfrac{\dfrac{\dfrac{}{\begin{array}{c}\{O, x, t, e \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon)\} \\ \text{e := }\mathsf{peer}(\text{E}) \\ \{O, x, t, e \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) \wedge e = \varepsilon\}\end{array}}\ \text{Lem. 7}}{}\ \textsc{Frame} \qquad (4)}{\begin{array}{c}\{O, x, t, e \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) * I_m(\varepsilon, x)\} \\ \text{e := }\mathsf{peer}(\text{E});\ \mathsf{new}(\text{m,e,x});\ \mathsf{log}^{?}(\text{m,e,x,t}); \\ \{O, x, t, e \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) * I_m(\varepsilon, x)\}\end{array}}\ \textsc{Seq}$$

(4)

$$
\dfrac{
\textsc{Weak}\ \dfrac{
\textsc{Frame}\ \dfrac{
\textsc{Ax}\ \dfrac{}{
\begin{array}{c}
\{O, x, e, t \Vdash e = \varepsilon \land x = v \land \mathsf{emp}\}\\
\mathtt{new}(\mathrm{m,e,x})\\
\{O, x, e, t \Vdash e = \varepsilon \land x = v \land I_m(\varepsilon, x)\}
\end{array}}
}{}
}{
\begin{array}{c}
\{O, x, e, t \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) \land e = \varepsilon\}\\
\mathtt{new}(\mathrm{m,e,x})\\
\{O, x, e, t \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) * I_m(\varepsilon, x) \land e = \varepsilon\}
\end{array}}
\qquad
\dfrac{
\dfrac{}{
\begin{array}{c}
\{O, x, e, t \Vdash I_m(\varepsilon, x) \land e = \varepsilon\}\\
\mathtt{log}^{?}(\mathrm{m,e,x,t})\\
\{O, x, e, t \Vdash I_m(\varepsilon, x) \land e = \varepsilon\}
\end{array}}\ \text{Lem. 7}
}{}\ \textsc{Frame}
}{
\begin{array}{c}
\{O, x, e, t \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) \land e = \varepsilon\}\\
\mathtt{new}(\mathrm{m,e,x});\ \mathtt{log}^{?}(\mathrm{m,e,x,t});\\
\{O, x, e, t \Vdash E \overset{ep}{\mapsto} (C[b], \varepsilon) * I_m(\varepsilon, x)\}
\end{array}}\ \textsc{Seq}
$$

We can finally prove the soundness of our proof system (Theorem 6).

Proof The result follows from the soundness of abstract separation logic. If $(\!|.|\!)$ is a semantics of atomic commands that satisfy their respective specifications, then any provable triple is valid with respect to these semantics. This is ensured by Lemma 5. However, abstract separation logic does not treat the `local x in` p construction. As it boils down to a mere sequence of actions $\mathrm{n}_x; p; \mathrm{d}_x$, it does preserve validity. □

B Proofs about copyless message passing

This appendix is dedicated to the proof of a generalization of Theorem 3 that requires the notion of *contract obedient states*.

B.1 Half-duplex channels

ε is coupled with ε' if $\hat{\sigma} \vDash \langle \tau \rangle \varepsilon \overset{ep}{\mapsto} \varepsilon' * \top$. If ℓ is a history, let $msg(\ell) \in MsgId^{\omega}$ denote the word formed by message identifiers in ℓ. If ε, ε' are coupled endpoints, we say that w transits from ε to ε', and write $\hat{k} : \varepsilon \xrightarrow{w} \varepsilon'$, if the write counter and history $n_!$, ℓ of ε and the read history and counter $n_?$, ℓ of ε' are such that $n_? \leq n_!$ and $w = \ell[n_? + 1..n_!]$, *i.e.* the (possibly empty) word $\ell[n_? + 1].\ell[n_? + 2] \ldots \ell[n_!]$. Note that an alternative, maybe more intuitive notation could be $\hat{k} : \varepsilon' \xleftarrow{w} \varepsilon$, as w is read in the opposite direction as the transmission in the queue, that is, the first letter of w is the first letter to be read by ε'.

If a, b are states of contract C, we write $C : a \xrightarrow{?w} b$ to denote that there is a read path from a to b labeled by w in C. Finally, if $\hat{k}(\varepsilon) = (-, a, -, -, -, -, -)$ we let $\mathrm{state}_{\hat{k}}(\varepsilon) \triangleq a$ be the contract's state of the endpoint.

We will say that a memory state is half-duplex if for any coupled endpoints, there are messages in transfer in only one of the two directions, but not both. Moreover, this sequence of unread messages hat to correspond to a read path in the follower's contract.

Definition 1 (Half-duplex) *A state $\hat{\sigma}$ is said to be* half-duplex *if there is an endpoint head $\hat{k}$ such that*

1. *$\hat{\sigma}$ is partitioned and $\hat{k}$ is the endpoint heap of $\mathrm{closure}(\hat{\sigma})$,*

2. *for all coupled endpoints* $\varepsilon, \varepsilon'$, *if* $\hat{k} : \varepsilon \xrightarrow{w} \varepsilon'$ *with* $|w| \neq 0$, *then* $\mathrm{contract}_{\hat{k}}(\varepsilon') : \mathrm{state}_{\hat{k}}(\varepsilon') \xrightarrow{?w} \mathrm{state}_{\hat{k}}(\varepsilon)$*;*
3. *for all coupled endpoints* $\varepsilon, \varepsilon'$, *there are* w, w' *such that* $\hat{k} : \varepsilon \xrightarrow{w} \varepsilon'$ *and* $\hat{k} : \varepsilon' \xrightarrow{w'} \varepsilon$, *and* $|w| = 0$ *or* $|w'| = 0$ *(possibly both).*

A crucial property of deterministic, positional contracts is that they are half-duplex.

Lemma 2 *For all half-duplex state* $\hat{\sigma}$, *for all atomic command* c *possibly ruled by some deterministic positional contract,* $(\!|c|\!)_g(\hat{\sigma})$ *is half-duplex.*

Proof Condition 1 comes from the definition of $(\!|.|\!)_g$. For other conditions, the proof depends on the atomic command involved. Let $\hat{\sigma}$ be some contract obedient state and $\hat{\sigma}' \in (\!|c|\!)(\hat{\sigma})$.

- If c is one of the pointer instructions, one may observe that the endpoints are unchanged, and the same for logged cells.
- If c is `open`: two endpoints are added and others are unchanged. The two new endpoints have dual histories and are in the same state, so conditions 2 and 3 are obviously satisfied.
- If c is `close`, two endpoints are removed and others are unchanged. So conditions 2 and 3 are obviously preserved.
- Assume c is a send over ε towards ε' with message identifier m under contract C, and let $\hat{\sigma}' = (\!|c|\!)_g(\hat{\sigma})$, $\hat{k}'$ is the endpoint heap in $\mathit{closure}(\hat{\sigma}')$ and $\hat{k}$ the queue in $\mathit{closure}(\hat{\sigma})$. Then for all coupled $\varepsilon_0, \varepsilon_1$ different from $\varepsilon, \varepsilon'$, the property $\varepsilon_0 \xrightarrow{w} \varepsilon_1$ holds in $\hat{k}$ if and only if it holds in $\hat{k}'$. Hence, to establish condition 2 and 3, we need only to check them for $\varepsilon \to \varepsilon'$ and $\varepsilon' \to \varepsilon$. Let w_1, w_2 be such that:
$$\hat{k} : \varepsilon \xrightarrow{w_1} \varepsilon' \quad \varepsilon' \xrightarrow{w_2} \varepsilon$$
$$\hat{k}' : \varepsilon \xrightarrow{w_1.m} \varepsilon' \quad \varepsilon' \xrightarrow{w_2} \varepsilon.$$
By condition 2 in $\hat{\sigma}$, there is a read path $\bar{C} : \mathrm{state}_{\hat{k}}(\varepsilon') \xrightarrow{?w_1} \mathrm{state}_{\hat{k}}(\varepsilon)$, and since the send (in particular `contract`!) is fault free, there is an edge $C : \mathrm{state}_{\hat{k}}(\varepsilon) \xrightarrow{!m} \mathrm{state}_{\hat{k}'}(\varepsilon)$. So $\bar{C} : \mathrm{state}_{\hat{k}'}(\varepsilon') = \mathrm{state}_{\hat{k}}(\varepsilon') \xrightarrow{?w_1.m} \mathrm{state}_{\hat{k}'}(\varepsilon)$, which establishes the first half of condition 2. Let us assume that $|w_2| \neq 0$. Then by condition 2 in $\hat{\sigma}$, there is a (non-empty) read path $C : \mathrm{state}_{\hat{k}}(\varepsilon) \xrightarrow{?w_2} \mathrm{state}_{\hat{k}}(\varepsilon')$, so in particular there is a read edge outgoing from $\mathrm{state}_{\hat{k}}(\varepsilon)$. Since there is also a write edge for the send to be allowed, this contradicts the fact that the contract is positional. So $|w_2| = 0$, which ends to establish 2 and 3.
- Assume c is a receive over ε from a send posted on ε', with message identifier m under contract C, and let $\hat{\sigma}' = (\!|c|\!)_g(\hat{\sigma})$, $\hat{k}'$ be the endpoint heap in $\mathit{closure}(\hat{\sigma}')$ and $\hat{k}$ the endpoint heap in $\mathit{closure}(\hat{\sigma})$. Then for all coupled $\varepsilon_0, \varepsilon_1$ different from $\varepsilon, \varepsilon'$, the property $\varepsilon_0 \xrightarrow{w} \varepsilon_1$ holds in $\hat{k}$ if and only if it holds in $\hat{k}'$. Hence, to establish condition 2 and 3, we need only to check them for $\varepsilon \to \varepsilon'$ and $\varepsilon' \to \varepsilon$. Let w_1, w_2 be such that:
$$\hat{k} : \varepsilon \xrightarrow{m.w_1} \varepsilon' \quad \varepsilon' \xrightarrow{w_2} \varepsilon$$
$$\hat{k}' : \varepsilon \xrightarrow{w_1} \varepsilon' \quad \varepsilon' \xrightarrow{w_2} \varepsilon$$

By condition 3 in $\hat{\sigma}$, $|w_2| = 0$ because $|m.w_1| > 0$, which shows condition 3 and half of condition 2 hold in $\hat{\sigma}'$. By condition 2 in $\hat{\sigma}$, there is a read path C : $\mathrm{state}_{\hat{k}}(\varepsilon) \xrightarrow{?m.w_1} \mathrm{state}_{\hat{k}}(\varepsilon')$, and since the receive (in particular `contract?`) is fault free, there is an edge C : $\mathrm{state}_{\hat{k}}(\varepsilon) \xrightarrow{?m} \mathrm{state}_{\hat{k}'}(\varepsilon)$. Let us assume by absurd that the path labeled by $?m.w_1$ does not go through $\mathrm{state}_{\hat{k}'}(\varepsilon)$. Then there are two edges outgoing from $\mathrm{state}_{\hat{k}}(\varepsilon)$ that are labeled with $?m$, which contradicts the fact that the contract is deterministic. So C : $\mathrm{state}_{\hat{k}}(\varepsilon) \xrightarrow{?m.w_1} \mathrm{state}_{\hat{k}}(\varepsilon')$ factors as $C : \mathrm{state}_{\hat{k}}(\varepsilon) \xrightarrow{?m} \mathrm{state}_{\hat{k}'}(\varepsilon) \xrightarrow{?w_1} \mathrm{state}_{\hat{k}}(\varepsilon') = \mathrm{state}_{\hat{k}'}(\varepsilon')$, which ends the proof. □

B.2 Contract obedient states

We are now ready to give the definition of a contract obedient state, that will play a central role in the proof of Theorem 3.

Let $SL(\hat{\sigma})$ denote, for a legal memory state $\hat{\sigma}$, the set of timestamps, polarized with !, that appear in the histories of any coupled endpoints between what has been received and what has been sent, that is $SL(\hat{\sigma}) \triangleq \bigcup\{ts(\ell[n_?..n_!])^! \mid closure(\hat{\sigma}) \vDash \varepsilon \overset{ep}{\mapsto}(-, \varepsilon', -, n_!, -, \ell) * \varepsilon' \overset{ep}{\mapsto}(-, \varepsilon, n_?, -, \ell, -) * \mathsf{True}\}$.

Definition 3 *A state $\hat{\sigma}$ is said to be* contract obedient *when it meets the following conditions:*

1. *$\hat{\sigma}$ is legal;*
2. *$\hat{\sigma}$ is half-duplex;*
3. *the unmatched logs are exactly the unmatched send logs, that is $UL(\hat{\sigma}) = SL(\hat{\sigma})$.*

To prove Theorem 3, we establish its generalization to contract obedient states:

Theorem 4 *For all provable program p with bounded contracts, for all $tr \in T(p)$, for all contract obedient state $\hat{\sigma}$, $[\![tr]\!]_{nt}(closure(\hat{\sigma})) = closure([\![tr]\!]_g(\hat{\sigma}))$.*

This result obviously entails Theorem 3 as $u = closure(u)$ is contract obedient. A key property of the proof of Theorem 4 is that contract obedience is preserved by commands:

Lemma 5 *For all legal, contract obedient state $\hat{\sigma}$, for all atomic command c ruled by a non-leaking contract, $(\!|c|\!)_g(\hat{\sigma})$ is contract obedient.*

Proof Condition 1 comes from the definition of $(\!|.|\!)_g$. Condition 2 comes from Lemma 2. For the last condition, the proof depends on the atomic command involved. Let $\hat{\sigma}$ be some contract obedient state and $\hat{\sigma}' \in (\!|c|\!)(\hat{\sigma})$.

- If c is one of the pointer instructions, one may observe that the endpoint cells are unchanged, and the same for logged cells.
- If c is `open`: two endpoints are added and others are unchanged, no logged cells are added, so condition 3 is preserved.
- If c is `send`, then logged cells tagged with $\tau^!$ are added, where τ is the timestamp consumed in the write history of ε, so condition 3 is preserved.

- If c is `receive`, then logged cells tagged with $\tau^?$ are added, where τ is the timestamp consumed in the read history values of ε. From the **DualMatch** property, the same logged cells tagged with $\tau^!$ were already in $\hat{\sigma}$, so this log becomes unmatched, and condition 3 is preserved.
- Finally, assume c is a `close` command. Then two endpoints are removed and others are unchanged, and logged cells are unchanged. So condition 3 is preserved provided these endpoints played no role in the definition of $SL(\hat{\sigma})$. This holds because the endpoints were in the same state in $\hat{\sigma}$, so as $\hat{\sigma}$ is half-duplex and the contract is bounded, they must have the same read and write histories in $\hat{\sigma}$, hence none of their logs are unmatched, and so condition 3 is preserved. □

B.3 Non-transferring actions

Definitions 6

- *We call a function f* non-transferring *if $transfer(f(\hat{\sigma})) = \{transfer(\hat{\sigma})\}$ for all $\hat{\sigma}$.*
- *We call* action *any atomic, sub-atomic, or trace-specific action.*
- *If $A \in \mathcal{P}(\hat{\Sigma}^{\mathsf{wf}})^{\top}$ and $\hat{\sigma} \in \hat{\Sigma}^{\mathsf{wf}}$, we simply write $A * \hat{\sigma}$ and $A - \hat{\sigma}$ to denote the set of states $\hat{\sigma}' \bullet \hat{\sigma}$ and $\hat{\sigma}' - \hat{\sigma}$ for all $\hat{\sigma}'$ in A.*

Lemma 7 *For all action c different from `log`†, `send` and `receive`, $(\!|c|\!)$ and $(\!|c|\!)_g$ are non-transferring.*

Proof None of the remaining actions modify the set of logged cells, hence the set of unmatched logs, nor do they modify the content of logged cells. □

Lemma 8 *For all action c different from `log`†, `send` and `receive`, for all contract obedient $\hat{\sigma}$,*

$$(\!|c|\!)_g(\hat{\sigma})\lfloor_{\mathbf{now}} = (\!|c|\!)(closure(\hat{\sigma})) - [\mathbf{now}]transfer(\hat{\sigma}) \ .$$

Proof We reason by case analysis:

- If c is non-allocating, that is c is neither `new` nor `open`, and $\mathrm{now}(\hat{\sigma}_1) = \emptyset$, then $(\!|c|\!)(\hat{\sigma} \bullet \hat{\sigma}_1) = (\!|c|\!)(\hat{\sigma}) * \hat{\sigma}_1$ (by locality and the fact that all $\hat{\sigma}$ in the footprint of $(\!|c|\!)$ form an antichain). Then

$$(\!|c|\!)(closure(\hat{\sigma})) = (\!|c|\!)(\hat{\sigma}\lfloor_{\mathbf{now}}) * [\mathbf{now}]transfer(\hat{\sigma})$$

  and $(\!|c|\!)(\hat{\sigma}\lfloor_{\mathbf{now}}) = (\!|c|\!)(\hat{\sigma})\lfloor_{\mathbf{now}}$, which ends the proof.
- If c is either one of the `new` or `open`, then $\hat{\sigma}' \in (\!|c|\!)(\hat{\sigma})$ is legal iff the new cells are not allocated at a location of a cell in transfer, that is if $(\hat{\sigma}' - \hat{\sigma}) \perp [\mathbf{now}]transfer((\!|c|\!)(\hat{\sigma})) = [\mathbf{now}]transfer(\hat{\sigma})$ (by Lemma 7). This exactly is what is achieved when letting $[\mathbf{now}]transfer(\hat{\sigma})$ frame in. □

B.4 Transferring actions

We now prove the base case of Theorem 4 for `send` and `receive`. This requires careful examination of the sub-atomic commands, so we begin by a technical observation.

Lemma 9 *For every contract obedient state $\hat{\sigma}$ and each sub-atomic command c different from* `new`*, $(\!|c|\!)_g(\hat{\sigma})$ (which is the same as $(\!|c|\!)_g^{nt}$ is either $\top$, $\emptyset$ or a singleton set.*

Proof Straightforward from their specifications. □

Lemma 10 *For all contract obedient state $\hat{\sigma}$,*

$$closure((\!|send(E.m,E')|\!)_g(\hat{\sigma})) = (\!|send(E.m,E')|\!)_g^{nt}(closure(\hat{\sigma})) \; .$$

Proof If $($ contract$^!$(E.m); enq(E,E'))$(\hat{\sigma})$ is $\emptyset$ or $\top$, then the equality holds trivially because both interpretations of `send(E.m,E')` start with these two actions. Hence, let us assume

$$[\![\text{contract}^!(\text{E.m}); \text{enq}(\text{E,E'})]\!](\hat{\sigma}) = \hat{\sigma}_1$$

and compute $[\![\text{send}(\text{E.m,E'})]\!]_g$ under this assumption. $(\text{d}_t;\ \text{t:= cur\_ts}^!(\text{E}))(\hat{\sigma}_1)$ cannot block, and cannot fault either because `contract(E.m)` has not. Hence let

$$\hat{\sigma}_2 = [\![\text{d}_t;\ \text{t:= cur\_ts}^!(\text{E})]\!](\hat{\sigma}_1) \; .$$

We have $\hat{\sigma}_2 = \hat{\sigma}_1 \bullet [t{\rightarrow}\tau]$, where $[t{\rightarrow}\tau]$ is the state where only the stack s is not empty, and $s(t) = \tau$.

The next command is `log`$^!$, which may fault if the invariant is not satisfied, *i.e.* iff

$$\hat{\sigma}_1 \nvDash (O \Vdash I_m(E,E')) * \mathsf{true}$$

Otherwise, let $\hat{\sigma}_m \leq \hat{\sigma}_1$ be such that $\hat{\sigma}_m \vDash O \Vdash I_m(E,E')$. There is only one such state because invariants of messages are precise. We have then

$$(\!|\text{log}^!(\text{m,E,E',t})|\!)(\hat{\sigma}_2) = \hat{\sigma}_2 \bullet [\tau^!]\hat{\sigma}_m \; .$$

This is well-defined because, according to **DisjointLogs** and the fact that $\hat{\sigma}$ is contract obedient hence legal, τ cannot already appear in $ts(\hat{\sigma}_2)$. Finally, $[\![\text{d}_t;\ \text{free}\,(\text{m,E,E'})]\!]$ does not fault, because `log`$^!$`(m,E,E',t)` has not, and, if we call $\hat{\sigma}_f$ the substate of $\hat{\sigma}_1$ such that $\hat{\sigma}_1 = \hat{\sigma}_f \bullet \hat{\sigma}_m$, we have

$$[\![\text{d}_t;\ \text{free}\,(\text{m,E,E'})]\!](\hat{\sigma}_2 \bullet [\tau^!]\hat{\sigma}_m) = [\![\text{free}\,(\text{m,E,E'})]\!](\hat{\sigma}_1 \bullet [\tau^!]\hat{\sigma}_m) = \hat{\sigma}_f \bullet [\tau^!]\hat{\sigma}_m \; .$$

Because $\hat{\sigma}$ is contract obedient, $\hat{\sigma}_f \bullet [\tau^!]\hat{\sigma}_m$ is, and $closure(\hat{\sigma}_f \bullet [\tau^!]\hat{\sigma}_m) = \hat{\sigma}_f \bullet \hat{\sigma}_m = \hat{\sigma}_1$.

Putting all this together gives us that either $\hat{\sigma}_1 \nvDash (O \Vdash I_m(E,E')) * \mathsf{true}$ and $[\![\text{send}(\text{E.m,E'})]\!]_g(\hat{\sigma}) = \top$, or $closure([\![\text{send}(\text{E.m,E'})]\!]_g(\hat{\sigma})) = \hat{\sigma}_1$.

Let us now show that either $\hat{\sigma}_1 \nvDash (O \Vdash I_m(E,E')) * \mathsf{true}$ and $[\![\text{send}(\text{E.m,E'})]\!]_g^{nt}(closure(\hat{\sigma})) = \top$, or $[\![\text{send}(\text{E.m,E'})]\!]_g^{nt}(closure(\hat{\sigma})) = \hat{\sigma}_1$, which will conclude the proof.

Because of the locality of sub-atomic commands, Lem. 9, and the fact that neither `contract` nor `enq` care about logged cells, we have

$$\begin{aligned}
&[\![\text{contract}^{!}(\text{E.m});\ \text{enq}(\text{E,E'})]\!](\mathit{closure}(\hat\sigma))\\
=\ &[\![\text{contract}^{!}(\text{E.m});\ \text{enq}(\text{E,E'})]\!](\hat\sigma\lfloor_{\mathbf{now}}) \bullet [\mathbf{now}]\mathit{transfer}(\hat\sigma)\\
=\ &\hat\sigma_1\lfloor_{\mathbf{now}} \bullet[\mathbf{now}]\hat\sigma_1\\
=\ &\mathit{closure}(\hat\sigma_1)
\end{aligned}$$

It is then easy to see that either $\hat\sigma_1 \nvDash (O \Vdash I_m(E,E'))*\mathsf{true}$ and $[\![\text{check\_inv}(\text{m,E,E'})]\!](\mathit{closure}(\hat\sigma_1)) = \top$, or $[\![\text{send}(\text{E.m,E'})]\!]_g^{nt}(\mathit{closure}(\hat\sigma)) = \hat\sigma_1$. □

Lemma 11 *For all contract obedient state $\hat\sigma$,*

$$\mathit{closure}((\!|x\ :=\ \mathit{receive}\,(E.m)|\!)_g(\hat\sigma)) = (\!|x\ :=\ \mathit{receive}\,(E.m)|\!)_g^{nt}(\mathit{closure}(\hat\sigma))\ .$$

Proof As for `send`, we only need to consider the case where $(\!|\,\text{contract}^{?}(\text{E.m});\ \text{x:=deq(E)}|\!)(\hat\sigma)$ does not fault nor loop, and produces a (single thanks to Lem. 9) state $\hat\sigma_1$. Moreover, because $[\![\text{x}\ :=\ \text{receive}\,(\text{E.m})]\!]_g^{nt}$ is precisely $(\!|\,\text{contract}^{?}(\text{E.m});\ \text{x:=deq(E)}|\!)$ and because these actions are local and are not affected by logged cells, we only need to prove that $\mathit{closure}([\![\text{x}\ :=\ \text{receive}\,(\text{E.m})]\!]_g^{nt}(\hat\sigma)) = \mathit{closure}(\hat\sigma_1)$.

Let
$$\hat\sigma_2 = [\![\text{n}_t;\ \text{n}_e;\ \text{t:= cur\_ts (E);\ e:=peer(E)}]\!](\hat\sigma_1)$$

The previous actions cannot fault because `contract`$^?$`(E.m)` has not. Following the notation of the previous proof, we have $\hat\sigma_2 = \hat\sigma_1 \bullet [t{\to}\tau, e{\to}\varepsilon]$ and let $\hat\sigma_1 = \hat\sigma_1' \bullet [x{\to}v]$ (x is indeed in the stack because `x:=deq(E.m)` did not fault. The next action adds all states satisfying $I_m(e,x)$ to $\hat\sigma_2$:

$$\begin{aligned}
S_3 &= (\!|\text{new(m,e,x)}|\!)(\hat\sigma_2)\\
&= \bigsqcap_{\substack{\hat\sigma' \le \hat\sigma_2\\ \hat\sigma' \vDash e,x \Vdash e=\varepsilon \wedge x=v \wedge \mathsf{emp}}} \{\hat\sigma_2 - \hat\sigma'\} * (e,x \Vdash e=\varepsilon \wedge x=v \wedge I_m(\varepsilon,v))\\
&= (\hat\sigma_1' \bullet [t{\to}\tau]) * (e,x \Vdash e=\varepsilon \wedge x=v \wedge I_m(\varepsilon,v))\\
&= (\hat\sigma_1' \bullet [x{\to}v, e{\to}\varepsilon, t{\to}\tau]) * I_m(\varepsilon,v)
\end{aligned}$$

Finally, we log the cells corresponding to $I_m(\varepsilon,v)$:

$$\begin{aligned}
S_4 &= (\!|\text{log(m,e,x,t)}|\!)((\hat\sigma_1' \bullet [x{\to}v, e{\to}\varepsilon, t{\to}\tau]) * I_m(\varepsilon,v))\\
&= (\!|\text{log(m,e,x,t)}|\!)([x{\to}v, e{\to}\varepsilon, t{\to}\tau] * I_m(\varepsilon,v)) * \hat\sigma_1'\\
&= \hat\sigma_1' * \langle \mathbf{now} * [\tau^{?}]\rangle[x{\to}v, e{\to}\varepsilon, t{\to}\tau] * I_m(\varepsilon,v)\\
&= \hat\sigma_1' * [x{\to}v, e{\to}\varepsilon, t{\to}\tau] * \langle \mathbf{now} * [\tau^{?}]\rangle I_m(\varepsilon,v)
\end{aligned}$$

Because $\hat\sigma$ is contract obedient, so is $\hat\sigma_1$, and because `x:=deq(E.m)` did not block, we now by **Asynch** that $\hat\sigma$, hence $\hat\sigma_1$ contain a substate $\hat\sigma_m$ that satisfies $I_m(\varepsilon,v)$. Because $\hat\sigma_1$ is partitioned and $[\tau^{!}]$ was unmatched, we know that $\hat\sigma_m$ was not allocated in $\hat\sigma_1$. This validates the equalities above, in particular the resulting set of states is not empty.

Removing e and t from the stack gives us next $\hat{\sigma}_1 * \langle \mathbf{now} * [\tau^?]\rangle I_m(\varepsilon, v)$. Now that all sub-atomic actions have been properly executed, we need to restrict these states to the set of legal ones. Using **DualMatch**, the only state that is left from reducing to legal states is $\hat{\sigma}_1 * \langle \mathbf{now} * [\tau^?]\rangle \hat{\sigma}_m$, and $[\tau^!]$ is now matched. We thus obtain

$$\mathit{closure}(\llbracket \mathrm{x} := \mathtt{receive}\,(\mathrm{E.m})\rrbracket_g^{nt}(\hat{\sigma})) = \mathit{closure}(\hat{\sigma}_1) \bullet \hat{\sigma}_m = \mathit{closure}(\hat{\sigma}_1)\ .$$

□

B.5 Properties of closed semantics

Lemma 12 *For all contract obedient state $\hat{\sigma}$, for all command c different from* `new`*,* `open`*,* `log`†*,* `send` *and* `receive`*,*

1. $(\!|c|\!)_g(\hat{\sigma}) = (\!|c|\!)(\hat{\sigma})$;
2. $\mathit{closure}((\!|c|\!)(\hat{\sigma})) = (\!|c|\!)(\mathit{closure}(\hat{\sigma}))$.

Proof All these commands are non-allocating, that is they satisfy that $\hat{\sigma}\perp\hat{\sigma}_1$ implies $(\!|c|\!)(\hat{\sigma})\perp\{\hat{\sigma}_1\}$ and $(\!|c|\!)(\hat{\sigma} \bullet \hat{\sigma}_1) = (\!|c|\!)(\hat{\sigma}) * \hat{\sigma}_1$ whenever $(\!|c|\!)(\hat{\sigma}) \neq \top$. Since $\hat{\sigma} \downharpoonright_{\mathbf{now}} \perp[\mathbf{now}]\mathit{transfer}(\hat{\sigma})$, we have $(\!|c|\!)(\hat{\sigma}\downharpoonright_{\mathbf{now}})\perp\{[\mathbf{now}]\mathit{transfer}(\hat{\sigma})\}$.

Since c is non-transferring, we have by Lemmas 7 and 8:

$$(\!|c|\!)(\hat{\sigma}) \downharpoonright_{\mathbf{now}} \perp\{[\mathbf{now}]\mathit{transfer}((\!|c|\!)(\hat{\sigma}))\}$$

which shows that all states in $(\!|c|\!)(\hat{\sigma})$ are partitioned, hence legal because $\hat{\sigma}$ is contract obedient, so $(\!|c|\!)(\hat{\sigma}) = (\!|c|\!)_g(\hat{\sigma})$. Moreover,

$$\begin{aligned}(\!|c|\!)(\mathit{closure}(\hat{\sigma})) &= (\!|c|\!)\big(\hat{\sigma} \downharpoonright_{\mathbf{now}} \bullet[\mathbf{now}]\mathit{transfer}(\hat{\sigma})\big)\\ &= (\!|c|\!)(\hat{\sigma}\downharpoonright_{\mathbf{now}}) * [\mathbf{now}]\mathit{transfer}(\hat{\sigma})\\ &= (\!|c|\!)(\hat{\sigma}\downharpoonright_{\mathbf{now}}) * [\mathbf{now}]\mathit{transfer}((\!|c|\!)(\hat{\sigma}))\\ &= (\!|c|\!)(\hat{\sigma}) \downharpoonright_{\mathbf{now}} *[\mathbf{now}]\mathit{transfer}((\!|c|\!)(\hat{\sigma}))\\ &= \mathit{closure}((\!|c|\!)(\hat{\sigma})) \qquad \square\end{aligned}$$

Lemma 13 *For all contract obedient state $\hat{\sigma}$, for all atomic command c,*

$$\mathit{closure}((\!|c|\!)_g(\hat{\sigma})) = (\!|c|\!)_g^{nt}(\mathit{closure}(\hat{\sigma}))\ .$$

Proof

– If c is neither `send` nor `receive` then, by Lemma 8, we have $((\!|c|\!)_g(\hat{\sigma}))\downharpoonright_{\mathbf{now}} = (\!|c|\!)(\mathit{closure}(\hat{\sigma})) - [\mathbf{now}]\mathit{transfer}(\hat{\sigma})$, and $\mathit{transfer}((\!|c|\!)_g(\hat{\sigma})) = \mathit{transfer}(\hat{\sigma})$ by Lemma 7, so

$$\begin{aligned}\mathit{closure}\big((\!|c|\!)_g(\hat{\sigma})\big) &= (\!|c|\!)_g(\hat{\sigma}) \downharpoonright_{\mathbf{now}} *[\mathbf{now}]\mathit{transfer}((\!|c|\!)_g(\hat{\sigma}))\\ &= (\!|c|\!)_g(\hat{\sigma}) \downharpoonright_{\mathbf{now}} *[\mathbf{now}]\mathit{transfer}(\hat{\sigma}) && \text{by Lemma 7}\\ &= ((\!|c|\!)(\mathit{closure}(\hat{\sigma})) - [\mathbf{now}]\mathit{transfer}(\hat{\sigma})) * [\mathbf{now}]\mathit{transfer}(\hat{\sigma})\\ & && \text{by Lemma 8}\\ &= (\!|c|\!)(\mathit{closure}(\hat{\sigma}))\\ &= (\!|c|\!)_g^{nt}(\mathit{closure}(\hat{\sigma}))\end{aligned}$$

– The cases of `send` and `receive` have been treated by Lemmas 10 and 11.

B.6 Proof of Theorem 4

We show by induction over tr that whenever $\hat{\sigma}$ is contract obedient, $[\![tr]\!]_g(\hat{\sigma})$ is contract obedient, and

$$closure([\![tr]\!]_g(\hat{\sigma})) = [\![tr]\!]_g^{nt}(closure(\hat{\sigma}))\ .$$

- The base case is proved by Lemmas 5 and 13.
- Then if $tr = tr_1; tr_2$, one has

$$\begin{aligned} closure([\![tr_1; tr_2]\!]_g(\hat{\sigma})) &= [\![tr_2]\!]_g^{nt}(closure([\![tr_1]\!]_g(\hat{\sigma}))) \\ &= [\![tr_2]\!]_g^{nt}([\![tr_1]\!]_g^{nt}(closure(\hat{\sigma}))) \\ &= [\![tr_1; tr_2]\!]_g^{nt}(closure(\hat{\sigma})) \end{aligned}$$

  which ends the proof. □