Sharing Contract-Obedient Endpoints

Étienne Lozes and Jules Villard

December 2011

Research report LSV-11-23

Laboratoire Spécification & Vérification

École Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

Sharing Contract-Obedient Endpoints
LSV Research Report

Étienne Lozes[1,2] Jules Villard[3]

[1] Universität Kassel, Germany
[2] LSV, ENS Cachan, CNRS, France
[3] Queen Mary University of London, UK

Abstract. Most of the existing verification techniques for programs based on message passing suppose either that channel endpoints are used in a linear fashion, where at most one thread can be considered as the owner of an endpoint at any given time, or that endpoints may be used arbitrarily by any number of threads. The former approach forbids the sharing of channels, while the latter limits what is provable about programs, since no constraint is put on the usage of channels. In this paper, we propose a midpoint between these techniques by extending a previously published proof system based on separation logic to allow the sharing of endpoints. We identify two independent mechanisms for supporting sharing: the standard technique based on reasoning with permissions, and a new technique based on what we call ownership on demand. We formalize these two techniques in a proof system, illustrate them on several examples, and we extend Villard's semantics [19] and soundness proofs to support sharing.

Introduction

In a recent work, we introduced a proof system based on separation logic for a model of copyless message-passing concurrency inspired by the Sing# programming language [20] The main idea of this proof system was to impose on channel's endpoints the same ownership discipline as the one usually put on cells: at every point in time, each endpoint is logically owned by a unique thread that is the only one allowed to use it for sending and receiving. This restriction facilitates the analysis of communications, and allows in particular to check whether they obey a certain communication contract specified by the programmer. Such an analysis permits then the detection of communications errors like unspecified receptions or message orphans. This strict ownership discipline however excludes many interesting concurrent programs, as the only programs that can be proved are confluent [8], *i.e.* deterministic up to commutation of non-conflicting concurrent events. Relaxing the ownership discipline is however not obvious. In the setting of session types [11], such a linear usage of channels is essential in order to ensure the subject reduction property, and relaxations of this principle are known to be problematic: if a channel needs to be shared among several parties, the best that can be done is to type it with an unrestricted session type [9], which basically allows to send a unique type of message for the rest of the session.

Consider for instance a voting protocol, where an authority sends a voting request to voters, and then collects their votes:

```
authority(voter,N) {
  local e,f,x;
  (e,f) = open(C);
  for i=0 to N-1 { send(vote_request,voter.(i),f) };
  for i=0 to N-1 { x = receive(vote,e); ...}
}
voter(ep) {
  local f,opinion;
  f = receive(vote_request,ep); ...
  send(vote,f,opinion);
}
```

In this code, the authority allocates a channel (e, f) for the vote, then gives access to f to all voters, and finally collects all votes on e. On the other side, each voter i receives f on its local endpoint ep (assumed to be paired with endpoint `voter.(i)`), and uses it to send its vote. The non-linear usage of endpoint `f` cannot be checked against a communication contract in our proof system, but it could be typed with an unrestricted type $C = un\ ?\mathrm{vote}; C$, where type C would guarantee that the session on channel (e, f) contains `vote` messages only. However, unrestricted types cannot ensure that exactly N messages will be sent, nor allow to distinguish among different kinds of messages, like `vote_yes` and `vote_no`.

Going back to Separation Logic, it could be expected to go beyond these limitations by introducing fractional ownership for endpoints, using the *permissions* mechanism [4,3]. In fact, such a mechanism has already been considered in other proof systems for message-passing concurrency based on Separation Logic [1,18,10]. Sharing contract-obedient endpoints with permissions is however problematic, as a consistent view of the contract's state of the endpoint should be maintained among all sharers. Sends and receives on a shared endpoint seem thus restricted to messages that do not change the contract's state, which does not permit significantly more than what is achieved with unrestricted types. Consider for instance the following code featuring a seller bargaining the price of its product, using an endpoint e, with several buyers concurrently accessing the peer enpoint f of e:

```
seller(e) {
  local price;
  price=0;
  while not good(price) {
    send(product_descr,e);
    price=receive(offer,e);
  }
}
buyer(f) {
  local x;
  receive(product_descr,f);
  x = think_about_it();
  send(offer,f,x)
}
```

The most natural contract that could be designed for the communication channel (e, f) is the one specifying that the sequence of message exchanges along (e, f) is !product_descr.?offer*. However, any contract expressing that property contains at least two states, which makes the above program unprovable using fractional permissions.

In this work, we introduce another relaxation to the ownership discipline that allows sharing endpoints. The main idea of this relaxation can be illustrated on the seller and buyers' example. The crucial observation for proving this example is to consider that all buyers have no assumption on the contract's state of f until they have received the product_descr message. Considering that, a buyer that has received a product_descr message is the only one allowed to make assumptions on the contract's state of f, it can safely change it without breaking someone else's asumption. But then, to do so, we should not require to own the (contract's state of the) endpoint f for a reception $receive(f, m)$ on f. Instead, we assume that the received message m grants *a posteriori* the ownership of (the contract's state of) f. In other words, a buyer is justified to be able to receive a product_descr message on f right after it has received this message.

This relaxation raises several questions. First, about its soundness, considering that it is not clear how contract obedience could still be guaranteed with this relaxation. Second, about its relevance for real programs, considering that this example is not necessarily representative of the traditional schemes of message-passing programming. Third, about the class of concurrent programs it supports, and how far this class differs from deterministic concurrent programs. We answer these questions as follows. First, we show that our relaxed proof system keeps all of the guarantees that we proved for its original version, including partial correctness, absence of races, unspecified receptions, or message orphans. Second, we show that it guarantees other properties that we did not completely formalize in previous works, namely contract obedience and absence of memory leaks. Third, we show that it provides a proof method for encoding many standard synchronization primitives like multicast, synchronization barriers, or locks. Fourth, it supports an encoding of the internal choice, which shows that all non-deterministic race-free programs can be proved.

Outline In the first section, we introduce a toy programming language and some problems of interest. In the second section, we introduce communication contracts. In the third section, we formalize our proof system. In the fourth section, we illustrate our proof system on several examples of non-confluent concurrent programs. In the fifth section, we provide an operational semantics for our toy programming language. In the sixfth section, we establish the soundness of our proof system with respect to this semantics. In the seventh section, we discuss some consequences of the soundness of our proof system on the safety of provable programs. We conclude with related works.

1 Background

1.1 A Model of Sing#

We consider a programming environment *à la* Sing#, where threads communicate via a global shared memory and rely on communication channels for synchronisation. Channels are pairs of FIFO buffers and always consist of exactly two endpoints. Each endpoint can be used to send to and receive from the other endpoint. A send instruction `send(m,e,v1,..,vn)` consists thus of an endpoint e, a message tag m, and values $v_1, .., v_n$ for the n parameters associated to the message tag m. A receive instruction specifies which message tag is expected; if different message tags could be expected, a switch construct can be used. Finally, endpoints are heap objects, and can be dynamically allocated by the instruction `(e,f)=open(C)` and disposed by `close(e,f)`. We equip our programming language with heap-manipulating primitives specialised for cells with two fields.

We assume infinite sets $\mathsf{Var} = \{e, f, x, y, \dots\}$, $\mathsf{Loc} = \{l, \dots\}$, $\mathsf{Endpoint} = \{\varepsilon, \dots\}$, $\Sigma = \{m, \dots\}$, $\mathsf{CIdent} = \{C, \dots\}$, $\mathsf{Control} = \{q, \dots\}$ and $\mathsf{Val} = \{v, \dots\}$ of respectively variables, memory locations, endpoints, message identifiers, contract identifiers, contract states and values. All sets but values are pairwise disjoint, and $\mathsf{Loc} \uplus \mathsf{Endpoint} \uplus \mathbb{N} \subseteq \mathsf{Val}$. The grammar of expressions, boolean expressions, atomic commands and programs is as follows:

$$E ::= x \in \mathsf{Var} \mid v \in \mathsf{Val} \qquad i ::= 0 \mid 1 \qquad B ::= E = E \mid B \text{ and } B \mid \text{not } B$$

$$\begin{aligned}
c &::= \text{assume(B)} \mid \text{x = E} \mid \text{x = new()} \mid \text{E.i = E} \mid \text{x = E.i} \mid \text{free(E)} \\
&\quad \mid \text{(e,f) = open(C)} \mid \text{close(E,E)} \mid \text{send(m, E,E1,..,En)} \\
&\quad \mid \text{(x1 ,.., xn)= receive(m, E)} \mid \text{switch}\{\mathit{cases}\} \\
\mathit{cases} &::= \sqcup \mid \text{case (x1 ,.., xn)=receive(m,E):}\{p\}\ \mathit{cases} \\
p &::= c \mid \text{skip} \mid p;\ p \mid p \parallel p \mid p + p \mid p^* \mid \text{local x in } p
\end{aligned}$$

`assume(B)` does nothing if B holds and blocks otherwise. `x=new()` allocates a new cell and store its address in lx, `free(x)` disposes the cell at address `x`, `E.i=E'` sets the the i field of the E cell to be equal to E', and `x=E.i` copies the content of the i field of the E cell into the variable `x`. Compound commands are standard and are in order sequential and parallel composition, internal choice, Kleene iteration and local variable creation. We leave the encodings of `while` loops and `if` statements to the attention of the reader, and will freely use them in our examples.

Example 1. The following program allocates a cell, a channel, exchanges the memory cell between two threads `put` and `get` by passing a message `cell`, and closes the channel.

```
main(){(e,f)=open();x=new(); {put(e,x)||get(f)}; close(e,f);}
put(e,x) {send(cell,e,x);}
get(f) {y=receive(cell,f); dispose(y);}
```

Example 2. Consider now the situation where the sender (`put`) non-deterministically chooses either to send the cell or to keep it. On the other side, the receiver (`get`) needs to be ready to receive any of the two messages `cell` and `no_cell`, and will thus rely on a `switch`/`case` construct. Consider moreover that `get` disposes its endpoint unilaterally after it has handled `put`'s message. Disposing endpoint unilaterally is not exactly what our model allows, and the receiver instead sends a `clos` message that informs the other party that communications should be interupted. This message carries the endpoint that `get` wants to dispose, and `put` is then in charge to dispose it together with its peer.

```
main(){(e,f)=open();x=new(); {put(e,x)||get(f)};}
put(e,x) {
  if (*) send(cell,e,x) else {send(nocell,e,x); dispose(x);};
  ff = receive(clos,e);
  close(e,ff);
}
get(f) {
  switch {
    case y=receive(cell,f) : {dispose(y);}
    case receive(nocell,f) : {skip;}
  }
  send(clos,f);
}
```

1.2 Some Problems of Interest

Programs of our model of Sing# may suffer from several runtime errors: first, all errors that occur in memory-manipulating programs, second, all errors that occur in shared-memory concurrency, and third, errors specific to message-passing. Let us review the ones we will focus on in this paper:

Memory violations They occur when a dangling pointer is dereferenced, modified, or disposed.

Races They occur when two threads simultaneously try to access a variable or a memory location in an unprotected way, and at least one of the accesses is a write access.

Memory leaks They occur when the memory state becomes such that no program's continuation can deallocate all channels and cells.

Orphan messages They occur when a channel is closed while messages are left in at least one of its buffers.

Deadlocks A program is in a deadlock configuration if all of its threads are blocked on receptions.

Unspecified receptions They occur when a thread enters a `switch`/`case` construct and the buffer of one of the endpoints it scans starts with a message whose tag is not listed as a possible case in the `switch`/`case`.

Note that these problems are sometimes related and not always mutually exclusive. For instance, the program

$$P_0 \triangleq \text{send}(\text{m1,e}) \;||\; \text{switch}\ \{\text{case}\ \text{receive}(\text{m2,f})\text{:}\{\}\}$$

may be considered both as deadlocking and as triggering an unspecified reception. Similarly, replacing `get(f)` by `skip` in Example 1 would cause both a memory leak and an orphan message. However, none of these problems fully overlap, and in particular:

- orphan messages are not a special case of memory leaks: it may be the case that lost messages carried no ownership (*e.g.* the message `nocell` above);
- unspecified receptions are not a special case of deadlocks: we will distinguish the program $P_1 \triangleq$ `send(m1,e);||receive(m2,f);` from the program P_0 above: the former is only deadlocking.[1] –provided no environment accesses `f` concurrently. Conversely, the program P`||receive(m1,f);send(m2,e)` triggers an unspecified reception, but does not deadlock if P does not either.

All of these safety properties will be formalized in Sec 7, once we will have provided an operational semantics for our toy programming language.

2 Contracts

Our aim is to design a proof system for Hoare triples $\{A\}\ p\ \{B\}$ based on Separation Logic that will prevent all of the aforementioned errors, except deadlocks[2]. Hoare triples are understood in terms of partial correctness and, for this reason, non-terminating and deadlocking programs generally have a proof. The other kinds of errors we mentioned are tractable in a standard proof system, but they require to model the communication contracts introduced by Sing#.

Suppose indeed that we have managed to derive a proof of a given program using our proof system. As usual in Separation Logic, this would guarantee that the program does not cause a memory violation nor a race. However, the other problems of interest that we listed are left unchecked. These problems are arguably out of the scope of the local reasoning approach: for detecting either an orphan message or an unspecified reception, one needs to be able to make some assumptions about the behaviour of the environment of a thread, in particular about the sequences of messages that it will send on those endpoints not owned by the program. This missing information is exactly the one conveyed by channel contracts.

2.1 Contracts as Automata

Channel contracts are finite automata that describe the protocol followed by the channel, *i.e.* which sequences of sends and receives are admissible on the

[1] In some cases, one may wish to authorize a thread to wait for a message in a buffer provided that other threads will pick all the other ones first. In order to support such behaviors, we distinguish reception from scanning, and consider that unspecified reception occurs only on scanning, not on reception.

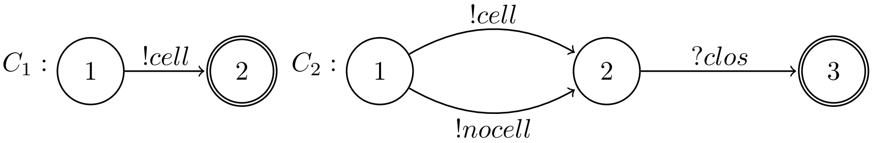
[2] The reason why we omit deadlocks is not that we find them uninteresting, but rather there is no simple way to treat them without introducing new notions; see Leino et al. [12], or session types [11] for analyses of message-passing programs dealing with deadlocks.

channel, and when the channel may be closed. A contract C is written from one of the endpoints' point-of-view, the other one following the dual contract $\bar{C}$, where sends ! and receives ? have been swapped. Contracts specify moreover an initial state, in which endpoints are assumed to be at channel creation, and a set of final states, in which endpoints have to be at channel closure.

Definition 1 (Contracts). *A contract is a tuple $C = (Q, \Sigma, \delta, q_0, F)$ such that Q is a finite set of states, Σ is a finite set of message tags, $\delta : Q \times \{!, ?\} \times \Sigma \to Q$ is a deterministic[3] transition function, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states.*

We write $\bar{C}$ for the dual of contract C, *i.e.* C where ! and ? are swapped, $\mathsf{init}(C)$ and $\mathsf{final}(C)$ for respectively q_0 and F, $\mathsf{succ}(C, q, \lambda)$ for the state q' (when it exists) such that $q \xrightarrow{\lambda} q' \in \Delta$, and $\mathsf{choice}(C, q)$ for the set of message identifiers m such that $\exists q'. q \xrightarrow{?m} q' \in \Delta$.

Example 3. The contracts for the two previous example programs are depicted below:



The first contract says that exactly one message `cell` will be sent from `e` to `f`, after which the channel may be closed. The second contract says that either `cell` or `nocell` will be sent, and this will be answered by a `clos` message, after which the channel may be closed.

2.2 Formal Semantics

Not all contracts will be validated by our analysis: their semantics should ensure the absence of unspecified receptions and orphan messages. For the sake of generality, we adopt here a more general semantics, and thus a broader notion of valid contracts as in other works[4]. We write Σ^* to denote the set of words over alphabet Σ, and $\llcorner\!\lrcorner$ to denote the empty word. A *configuration* of a contract C is a tuple

$$\gamma = (q, q', w, w') \in Q \times Q \times \Sigma^* \times \Sigma^*$$

where q is the current contract's state, q' is the dual contract state, and w, w' are the buffers' contents in both directions.

The semantics of a contract is provided by the transition system of a communicating finite state machine (CFSM):

[3] We could allow non-deterministic automata as well, but this would complicate the proof rules in Separation Logic a little bit. Moreover, determinism can always be assumed by standard automaton determinization.

[4] In particular, full-duplex communications are allowed: we do not restrict contracts to be positional [20], aka "autonomous" [17]

Definition 2 (Transition System). *We write*

$$(q_1^0, q_1^1, w_1^0, w_1^1) \to (q_2^0, q_2^1, w_2^0, w_2^1)$$

if and only if there is $i \in \{0,1\}$, $\lambda \in \{!,?\} \times \Sigma$ and a transition $q_1^i \xrightarrow{\lambda} q_2^i$ of the contract C_i (where $C_0 = C$ and $C_1 = \overline{C}$) such that

- *$q_1^{1-i} = q_2^{1-i}$;*
- *if $\lambda = !m$, then $w_2^{1-i} = w_1^{1-i}.m$ and $w_2^i = w_1^i$;*
- *if $\lambda = ?m$, then $w_1^i = m.w_2^i$ and $w_2^{1-i} = w_1^{1-i}$.*

For instance, the transition system of the second contract of Example 3 is

$$\begin{tikzcd}
 & 2,1,\text{cell},\sqcup \arrow[dr] & & & \\
1,1,\sqcup,\sqcup \arrow[ur] \arrow[dr] & & 2,2,\sqcup,\sqcup \arrow[r] & 2,3,\sqcup,\text{clos} \arrow[r] & 3,3,\sqcup,\sqcup \\
 & 2,1,\text{nocell},\sqcup \arrow[ur] & & &
\end{tikzcd}$$

Note that, due to the absence of bounds on the size of the communication buffers, the transition system associated to a contract may sometimes be infinite.

Definition 3 (Error Configurations). *Let C_0, C_1 be two contracts such that $C_1 = \overline{C})$. A configuration $\gamma = (q_0, q_1, w_0, w_1)$ of C_0 is called a message leak if $q_0 = q_1 \in F$ and $w.w' \neq \sqcup$, and an unspecified reception if there is $i \in \{0,1\}$ such that $w_i = m.w_i'$ for some w_i', and $m \notin \mathsf{choices}(C_i, q_i)$.*

Definition 4 (Valid Contracts). *A contract C is* valid *if all reachable configurations are neither message leaks nor unspecified receptions.*

In a previous work, we proposed sufficient conditions ensuring that a contract is valid. A contract is said *positional* if there are no q, q_1, q_2, a_1, a_2 such that $q \xrightarrow{!a_1} q_1$ and $q \xrightarrow{?a_2} q_2$. A contract that is both deterministic and positional prevents unspecified receptions [19]. They are moreover *realisable* in a certain sense, as shown by Stengel and Bultan [17]. A state is said *synchronizing* if all cycles going through this state contain at least a send and a receive transition. A contract that is deterministic, positional, and whose final states are synchronizing, is a valid contract [19]. Moreover, if all states are synchronizing, then the transition system is ensured to be finite, since communication buffers are then bounded by the length of the largest elementary cycle of the contract.

All the example contracts presented in this work are valid contracts for these reasons.

3 Separation Logic

Being based on Separation Logic, our proof system will ensure that every proved program follows some locality principles. If we were to consider a proof system without sharing, these locality principles could be summarized as follows:

Ownership hypothesis Each program component (procedure or thread) owns a subpart of the heap: it only reads and update that part of the heap.

Separation property At any point in the execution, each cell is owned by either exactly one thread, or by a message stored in in a queue.

Example 4. In the program of Ex. 1, `put` and `get` own disjoint parts of the heap: initially, `put` owns the endpoint `e` and the cell `x`, whereas `get` owns the endpoint `f`. During execution, the ownership of `x` is transferred from `put` to the message `cell`, and ultimately to `get`, which justifies its disposal by `get`.

3.1 Assertions: Overview

State assertions convey information on the part of the heap that is owned by the thread it refers to. Basic state assertions should thus describe the ownership of a cell or an endpoint. For cells, we simply adopt the usual predicate $x \mapsto (v_1, v_2)$ that specifies that the cell at address x is allocated, contains v_1 in the first field, and v_2 in the second one. For endpoints, we introduce a new predicate $e \overset{ep}{\mapsto} C\langle a\rangle$ that accounts for the ownership of an endpoint e ruled by a contract C, and currently in a state a of C.

Example 5. The example code below features the annotations that formalizes the explanations given in Example 4. Separation formulas appear in square brackets: $x \mapsto -$ denotes the ownership of a cell x, `emp` indicates that nothing is known to be allocated (or, alternatively, that nothing in the heap is owned at this program's point), and the separating conjunction $*$ adds up disjoint pieces of owned heap.

```
message cell (x) [$x \mapsto -$]
contract C {
{  initial state a {!cell->b;};
  final state b {};
}
main() [emp] {
  (e,f)=open(C);x=new(); {put(e,x);||get(f);}; close(e,f);
} [emp]
put(e,x) [$e \overset{ep}{\mapsto} C\langle a\rangle * x \mapsto -$] {send(cell,e,x);} [$e \overset{ep}{\mapsto} C\langle b\rangle$]
get(f) [$f \overset{ep}{\mapsto} \overline{C}\langle a\rangle$] {y=receive(cell,f);dispose(y);} [$f \overset{ep}{\mapsto} \overline{C}\langle b\rangle$]
```

The contract described in the code is just the contract C_1 of Example 3, in a Sing#-like syntax.

3.2 Assertions: Formal Definition

We consider Boyland's model of fractional permissions $\mathit{\Pi} \triangleq (0, 1]$ [4], essentially for simplicity, but any other permission model could be used instead in the following. Let us recall that permissions are equipped with a partial composition $+$, that a permission of 1 is the "total" or "write" permission, and a permission $\pi < 1$ is called a "partial" or "read" permission.

The set of formulas is as follows, assuming an additional set of logical variables $\mathsf{LVar} \triangleq \{X, Y, \dots\}$:[5]

$$\begin{array}{rll} A ::= & \mathsf{emp}_s \mid \mathsf{own}_\pi(x) \mid E = E & \text{stack predicates} \\ & \mid \mathsf{emp}_h \mid E \mapsto_\pi (E, E) \mid \dots & \text{heap predicates} \\ & \mid \mathsf{emp}_{ep} \mid E \overset{ep}{\mapsto} (C_\pi \langle . \rangle, E) \mid E \overset{ep}{\mapsto} (C_\pi \langle q^\pi \rangle, E) & \text{buffer predicates} \\ & \mid \neg A \mid A \wedge A \mid \exists X.\, A \mid A * A & \text{connectives} \end{array}$$

Intuitively, $E \overset{ep}{\mapsto} (C_\pi \langle . \rangle, E')$ represents the partial ownership of an endpoint E with contract C and peer E', but no ownership of its state, and $E \overset{ep}{\mapsto} (C_\pi \langle q^{\pi'} \rangle, E')$ the same endpoint with a partial ownership π' of its state. For simplicity, we will not use the first predicate in any examples, and we will write $E \overset{ep}{\mapsto}_\pi (C \langle q \rangle, E')$ for $E \overset{ep}{\mapsto} (C_\pi \langle q^\pi \rangle, E')$. As usual, we omit the permission subscript to mean that it is 1, and write emp for $\mathsf{emp}_h \wedge \mathsf{emp}_{ep}$, $E \mapsto (-, -)$ for $\exists X, Y.\, E \mapsto (X, Y)$. Free logical variables are existentially quantified. When $O = x_1, \dots, x_n$, we write as usual $O \Vdash A$ as a shorthand for $(\mathsf{own}(x_1) * \dots * \mathsf{own}(x_n)) \wedge A$. In some examples, we will omit part of the formula for conciseness: the $O \Vdash \dots$ part of the formula, the wildcard, or the exact peer of an endpoint are often left to the attention of the reader.

We will assume a fixed environment Γ that associates contracts to contract identifiers, and precise formulas to message identifiers. The footprint formulas (those annotating message identifiers) may use the special variable src that will be interpreted as the endpoint that sends the message. We will often write $I_m(\mathsf{src}, \overrightarrow{x})$ for the footprint formula of m, where $\overrightarrow{x}$ are the message parameters.

3.3 Local states

We give a forcing semantics to the previous formula with respect to a memory state model that does not model queue contents, and will be later enriched for providing an operational semantics to our toy programming language.

Local states are tuples $(s, h, \dot{k})$ of $\mathsf{St\dot{a}te}$:

$$\mathsf{St\dot{a}te} \triangleq \mathsf{Stack} \times \mathsf{Heap} \times \mathsf{IQueue} \qquad \mathsf{Stack} \triangleq \mathsf{Var} \rightharpoonup_{\mathit{fin}} \Pi \times \mathsf{Val}$$

$$\mathsf{Heap} \triangleq \mathsf{Cell} \rightharpoonup_{\mathit{fin}} \Pi \times \mathsf{Val} \times \mathsf{Val}$$

$$\mathsf{IQueue} \triangleq \mathsf{Endpoint} \rightharpoonup_{\mathit{fin}} \mathsf{Endpoint} \times \Pi \times \mathsf{Ctt} \times (\{.\} + \Pi \times \mathsf{Control})$$

We define the peer function $\mathsf{mate}(\dot{\sigma}) : \mathsf{Endpoint} \rightharpoonup \mathsf{Endpoint}$, for $\dot{\sigma} = (s, h, \dot{k})$, as the function with the same domain as $\dot{k}$ such that $\mathsf{mate}(\dot{k})(\varepsilon) = \varepsilon'$ if $\dot{k}(\varepsilon) = (\varepsilon', -, -, -)$. We define the functions $\mathsf{contract}(\dot{\sigma})$ and $\mathsf{cstate}(\dot{\sigma})$ similarly. We only consider well-formed states, namely states $\dot{\sigma} = (s, h, \dot{k})$ for which:

[5] For conciseness, we do not introduce recursive predicates for the heap as we do not need them in our examples, but they would not cause any problem for soundness. Some of them, like lists, trees, or doubly-linked lists are supported by Heap-Hop.

- For all allocated ε, if $\mathsf{mate}(\dot{\sigma}) = \varepsilon'$ is allocated, then $\mathsf{contract}(\dot{\sigma})(\varepsilon)$ and $\mathsf{contract}(\dot{\sigma})(\varepsilon')$ are dual.
- There is a total function $f : \mathsf{Endpoint} \to \mathsf{Endpoint}$ extending $\mathsf{mate}(\dot{\sigma})$ such that $f^2 = \mathsf{Id}_{\mathsf{Endpoint}}$ and $f(\varepsilon) \neq \varepsilon$ for all $\varepsilon \in \mathsf{Endpoint}$.

Two local states $\dot{\sigma} = (s, h, \dot{k})$ and $\dot{\sigma}' = (s', h', \dot{k}')$ are orthogonal, written $\dot{\sigma} \perp \dot{\sigma}'$, if:

- for all $x \in dom(s) \cap dom(s')$, if $s(x) = (\pi, v)$ and $s'(x) = (\pi', v')$, then $\pi + \pi' \leq 1$ and $v = v'$;
- for all $l \in dom(h) \cap dom(h')$, if $h(l) = (\pi, v_1, v_2)$ and $h'(l) = (\pi', v_1', v_2')$, then $\pi + \pi' \leq 1$, $v_1 = v_1'$, and $v_2 = v_2'$;
- for all $\varepsilon \in dom(\dot{k}) \cap dom(\dot{k}')$, $\mathsf{mate}(\dot{\sigma})(\varepsilon) = \mathsf{mate}(\dot{\sigma}')(\varepsilon)$, $\mathsf{contract}(\dot{\sigma})(\varepsilon) = \mathsf{contract}(\dot{\sigma}')(\varepsilon)$, permissions can be added, and if both defined, $\mathsf{cstate}(\dot{\sigma})(\varepsilon)$ and $\mathsf{cstate}(\dot{\sigma}')(\varepsilon)$ are equal, with permission sum ≤ 1.

Two orthogonal local states $\dot{\sigma}, \dot{\sigma}'$ can be composed to form a local state $\dot{\sigma}_1 \bullet \dot{\sigma}_2$ in the usual way. For instance, the store s of this composition is defined by

- $s(x) = s_1(x)$ if $x \in dom(s_1) \setminus dom(s_2)$,
- $s(x) = s_2(x)$ if $x \in dom(s_2) \setminus dom(s_1)$,
- $s(x) = (\pi_1 + \pi_2, v)$ if $x \in dom(s_2) \cap dom(s_1)$, $s_1(x) = (\pi_1, v)$ and $s_2(x) = (p_2, v)$
- $s(x)$ is undefined otherwise

Local states equipped with $\bullet$ form a separation algebra [5], *i.e.* a commutative cancellative partial monoid, with unit $\dot{u} = (\emptyset, \emptyset, \emptyset)$. The restriction of this algebra to well-formed states (in particular, two states are orthogonal only if their composition is well-formed), is still a separation algebra.

Given two local states $\dot{\sigma}_1$ and $\dot{\sigma}_2$, $\dot{\sigma}_1$ is a substate of $\dot{\sigma}_2$, written $\dot{\sigma}_1 \preceq \dot{\sigma}_2$, if there is $\dot{\sigma}$ such that $\dot{\sigma}_1 \bullet \dot{\sigma} = \dot{\sigma}_2$. In this case, $\dot{\sigma}$ is unique (as $\dot{\mathsf{State}}$ is a separation algebra), thus we can write $\dot{\sigma}_2 - \dot{\sigma}_1$ for the only such $\dot{\sigma}$. The binary relation $\preceq$ is a partial order which enjoys the following property: any bounded, increasing sequence $(\dot{\sigma}_i)_{i \geq 0}$ admits a lub, and $\bullet$ is a continuous operator.

Finally, we interpret formulas against well-formed local states. Let $[\![E]\!]_s$ denote the semantics of E with respect to s, *i.e.* $[\![x]\!]_s = s(x)$ and $[\![v]\!]_s = v$.

$$
\begin{array}{lll}
(s, h, \dot{k}) \vDash \mathsf{emp}_s & \text{iff} & s = \emptyset \\
(s, h, \dot{k}) \vDash \mathsf{own}_\pi(x) & \text{iff} & s(x) = (\pi, -) \\
(s, h, \dot{k}) \vDash E_1 = E_2 & \text{iff} & [\![E_1]\!]_s = [\![E_2]\!]_s \\
(s, h, \dot{k}) \vDash \mathsf{emp}_h & \text{iff} & h = \emptyset \\
(s, h, \dot{k}) \vDash E \mapsto_\pi (E_0, E_1) & \text{iff} & h([\![E]\!]_s) = (\pi, [\![E_1]\!]_s, [\![E_2]\!]_s) \\
(s, h, \dot{k}) \vDash \mathsf{emp}_{ep} & \text{iff} & \dot{k} = \emptyset \\
(s, h, \dot{k}) \vDash E \overset{ep}{\mapsto} (C_\pi\langle . \rangle, E') & \text{iff} & \dot{k}([\![E]\!]_s) = (\pi, C, [\![E']\!]_s, .) \\
(s, h, \dot{k}) \vDash E \overset{ep}{\mapsto} (C_\pi\langle q^{\pi'} \rangle, E') & \text{iff} & \dot{k}([\![E]\!]_s) = (\pi, C, [\![E']\!]_s, \pi', q) \\
\dot{\sigma} \vDash \neg A & \text{iff} & \dot{\sigma} \not\vDash A \\
\dot{\sigma} \vDash A_1 \wedge A_2 & \text{iff} & \dot{\sigma} \vDash A_1 \text{ and } \dot{\sigma} \vDash A_2 \\
\dot{\sigma} \vDash A_1 * A_2 & \text{iff} & \text{there are } \dot{\sigma}_1, \dot{\sigma}_2 \text{ such that} \\
 & & \dot{\sigma} = \dot{\sigma}_1 \bullet \dot{\sigma}_2,\ \dot{\sigma}_1 \vDash A_1,\ \text{and } \dot{\sigma}_2 \vDash A_2
\end{array}
$$

We say that A entails B, and write $A \vdash B$ if, for all well-formed local state $(s, h, \dot{k})$, if $(s, h, \dot{k}) \vDash A$, then $(s, h, \dot{k}) \vDash B$.

3.4 Proof Rules

Let us briefly review all standard proof rules of Separation Logic, recalled on Fig. 1.

- Skip: The program has already reached its final state. Using the frame rule, one can derive $\{A\}$ skip $\{A\}$ for any A.
- Assume: If the test is successful, the program terminates. The stack must contain the variables involved in the test.
- Assign: If the expression E can be read and variable x can be written, the value of x is updated to be the same as E.
- Lookup and Mutate: Similar, except that E (resp. E_1) must point to an allocated location.
- New and Dispose: Change an empty heap into a singleton heap and vice-versa.
- Sequence: This is the classical Floyd-Hoare rule for composing programs sequentially: the post-condition of the first program must be a valid precondition of the second one.
- Parallel: The rule for parallel composition accounts for *disjoint* concurrency: one has to be able to partition the heap into two disjoint portions that are valid respective preconditions for each of the two threads. The resulting post-conditions are glued together to form the post-condition of their parallel composition.
- Choice and Star are standard.
- Local: The rule allocates a new variable z which is used to prove p. The variable must be tracked throughout the proof of p and still be present at the end, when it is disposed.
- Frame This rule states that, whenever the execution of a program from a certain heap does not produce memory faults, it will not produce memory faults from a bigger heap either (a property called *safety monotonicity*), and the extra piece of heap will remain untouched by the program throughout its execution (a property called *locality*). With the frame rule, one can restrict the specification of programs to the cells they actually access (their *footprint* [16]). This also justifies that we can give the axioms for atomic commands in a very minimalistic way.
- Weakening is the standard Floyd-Hoare rule, whose soundness follows directly from the definition of what a valid Hoare triple is.
- Conjunction, Disjunction, and Existential are straightforward.

Let us now review the new rules supporting message-passing primitives.

Channel allocation and deallocation The rules for allocation and disposal are quite symmetric: `open` produces two fully owned, peered endpoints, whereas `close`

$$\textsc{Skip}\qquad \frac{}{\{\mathsf{emp}\}\ \mathrm{skip}\ \{\mathsf{emp}\}}$$

$$\textsc{Assume}\qquad \frac{}{\{\mathit{var}(B)_\pi \Vdash \mathsf{emp}\}\ \mathrm{assume}(B)\ \{\mathit{var}(B)_\pi \Vdash B \wedge \mathsf{emp}\}}$$

$$\textsc{Assign}\qquad \{\mathrm{x}, \mathit{var}(E)_\pi \Vdash E = X \wedge \mathsf{emp}\}\ \mathrm{x}\ {=}E\ \{\mathrm{x}, \mathit{var}(E)_\pi \Vdash \mathrm{x} = X \wedge \mathsf{emp}\}$$

$$\textsc{Lookup}\qquad \{\mathrm{x}, \mathit{var}(E)_\pi \Vdash E = Y \wedge Y \mapsto (X_0, X_1)\}\ \mathrm{x}\ {=}E.i\ \{\mathrm{x}, \mathit{var}(E)_\pi \Vdash Y \mapsto (X_0, X_1) \wedge \mathrm{x} = X_i\}$$

$$\textsc{Mutate}\qquad \{\mathit{var}(E_1, E_2)_\pi \Vdash E_1 \mapsto (-,-) \wedge E_2 = X_i\}\ E_1.i{=}E_2\ \{\mathit{var}(E_1, E_2)_\pi \Vdash E_1 \mapsto (X_0, X_1)\}$$

$$\textsc{New}\qquad \{\mathrm{x} \Vdash \mathsf{emp}\}\ \mathrm{x}\ =\ \mathrm{new}()\ \{\mathrm{x} \Vdash \mathrm{x} \mapsto (-,-)\}$$

$$\textsc{Dispose}\qquad \{\mathit{var}(E)_\pi \Vdash E \mapsto (-,-)\}\ \mathrm{dispose}(E)\ \{\mathit{var}(E)_\pi \Vdash \mathsf{emp}\}$$

$$\textsc{Sequence}\qquad \frac{\{A\}\ p\ \{A'\} \qquad \{A'\}\ p'\ \{B\}}{\{A\}\ p;p'\ \{B\}} \qquad\qquad \textsc{Parallel}\qquad \frac{\{A\}\ p\ \{B\} \qquad \{A'\}\ p'\ \{B'\}}{\{A * A'\}\ p\,||\,p'\ \{B * B'\}}$$

$$\textsc{Choice}\qquad \frac{\{A\}\ p\ \{B\} \qquad \{A\}\ p'\ \{B\}}{\{A\}\ p + p'\ \{B\}} \qquad\qquad \textsc{Star}\qquad \frac{\{A\}\ p\ \{A\}}{\{A\}\ p^*\ \{A\}}$$

$$\textsc{Local}\qquad \frac{\{\mathsf{own}(\mathrm{z}) * A\}\ p[\mathrm{x}{\leftarrow}\mathrm{z}]\ \{\mathsf{own}(\mathrm{z}) * B\}}{\{A\}\ \mathrm{local}\ \ \mathrm{x}\ \ \mathrm{in}\ p\ \{B\}}\ \mathrm{z} \notin \mathit{freevar}(A, p, B)$$

$$\textsc{Frame}\qquad \frac{\{A\}\ p\ \{A'\}}{\{A * B\}\ p\ \{A' * B\}} \qquad\qquad \textsc{Weakening}\qquad \frac{A' \Rightarrow A \qquad \{A\}\ p\ \{B\} \qquad B \Rightarrow B'}{\{A'\}\ p\ \{B'\}}$$

$$\textsc{Conjunction}\qquad \frac{\{A_1\}\ p\ \{B_1\} \qquad \{A_2\}\ p\ \{B_2\}}{\{A_1 \wedge A_2\}\ p\ \{B_1 \wedge B_2\}} \qquad\qquad \textsc{Disjunction}\qquad \frac{\{A_1\}\ p\ \{B_1\} \qquad \{A_2\}\ p\ \{B_2\}}{\{A_1 \vee A_2\}\ p\ \{B_1 \vee B_2\}}$$

$$\textsc{Existential}\qquad \frac{\{A\}\ p\ \{B\}}{\{\exists X.\, A\}\ p\ \{\exists X.\, B\}}$$

Figure 1: Standard proof rules.

consumes them. The former guarantees that they are in the initial state, while the latter requires them to be in a same final state.

$$\textsc{Open}\quad \frac{q_0 = \mathsf{init}(C)}{\{e, f \Vdash \mathsf{emp}\}\ \mathrm{(e,f)} = \mathrm{open(C)}\ \{e, f \Vdash e \overset{ep}{\mapsto}(C\langle q_0\rangle, f) * f \overset{ep}{\mapsto}(\overline{C}\langle q_0\rangle, e)\}}$$

$$\textsc{Close}\quad \frac{q \in \mathsf{final}(C)}{\{var(E, E')_\pi \Vdash E \overset{ep}{\mapsto}(C\langle q\rangle, E') * E' \overset{ep}{\mapsto}(\overline{C}\langle q\rangle, E)\}\ \mathrm{close\,(E,E')}\ \{var(E, E')_\pi \Vdash \mathsf{emp}\}}$$

Send and receive The rules for sending and receiving are also quite symmetric, but slightly more complicated. Send and receives indeed perform to changes on the owned heap: one the one hand they update the endpoint's state, and on the other hand, they acquire or release the ownership on the footprint of the message they deal with. If we do not want to consider reflexive ownership transfers, these two operations commute, as they touch disjoint part of the heap. But for supporting reflexive ownership transferring, we need to fix some order between both operations, and treat them with distinct rules.

To solve this issue, we use a pseudo-instruction $\mathrm{skip}_{E\lambda,E'}$, whose role is to change the contract's state of `E`, whose peer is at the address `E'`, with respect to the action λ, and to check that the transition is indeed authorized by the contract. Updating a contract state requires either a partial permission if the state is left unchanged, or a total permission otherwise. The pseudo instruction $\mathrm{skip}_{E\lambda,E'}$ is triggered by another rule, whose role is to account for the ownership transfer of the message footprint.

$$\textsc{Send}\quad \frac{\{var(E, \overrightarrow{F})_\pi \Vdash A\}\ \mathrm{skip}_{E!m,E'}\ \{var(E, \overrightarrow{F})_\pi \Vdash B * I_m(E, \overrightarrow{F})\}}{\{var(E, \overrightarrow{F})_\pi \Vdash A\}\ \mathrm{send(m,E,F1,..,Fn)}\ \{var(E, \overrightarrow{F})_\pi \Vdash B\}}$$

$$\textsc{Receive}\quad \frac{\{\overrightarrow{x}, var(E)_\pi \Vdash A * I_m(E', \overrightarrow{x})\}\ \mathrm{skip}_{E?m,E'}\ \{\overrightarrow{x}, var(E)_\pi \Vdash B\}}{\{\overrightarrow{x}, var(E)_\pi \Vdash A\}\ \mathrm{(x1\ ,..,\ xn)} = \mathrm{receive(m,E)}\ \{\overrightarrow{x}, var(E)_\pi \Vdash B\}}$$

$$\textsc{SkipLabel}\quad \frac{\mathsf{succ}(C, q, m) = q' \qquad q = q' \vee \pi = 1 \qquad O = var(E, E')_\pi}{\{O \Vdash E \overset{ep}{\mapsto}(C_{\pi'}\langle q^\pi\rangle, E')\}\ \mathrm{skip}_{E\lambda,E'}\ \{O \Vdash E \overset{ep}{\mapsto}(C_{\pi'}\langle q'^\pi\rangle, E')\}}$$

Note that, in the `send` case, the footprint of the message is transferred *after* the endpoint's state has been updated, and that it may contain the endpoint itself, and that conversely, in the `receive` case, the footprint of the message is transferred *before* the endpoint's state is checked and updated, and that the footprint may also contain the endpoint itself.

Switch/Case Finally, two new rules address the `switch`/`case` construct: the first rule dispatches switches on different endpoints in different subproofs, the second addresses `switch`/`case` on a unique endpoint only: in that case, this endpoint must be owned (at least partially), and the switch is checked to be exhaustive with respect to the possible incoming messages at the contract's state.

$$\textsc{SwitchDispatch}\quad \frac{\{A\}\ \mathsf{switch}\ \text{cases1}\ \{B\} \qquad \{A\}\ \mathsf{switch}\ \text{cases2}\ \{B\}}{\{A\}\ \mathsf{switch}\ \text{cases1}, \text{cases2}\ \{B\}}$$

$$\textsc{SwitchExhaust}\quad \frac{\begin{array}{c}\mathsf{choices}(C, q) \subseteq \{m_1, \ldots, m_n\} \\ \{E \overset{ep}{\underset{\pi}{\mapsto}} (C\langle q\rangle, E') * A\}\ \overrightarrow{x_i} = \mathsf{receive}(m_i, E); p_i\ \{B\}\ \text{for all}\ i\end{array}}{\{E \overset{ep}{\underset{\pi}{\mapsto}} (C\langle q\rangle, E') * A\}\ \mathsf{switch}\{\ldots, \overrightarrow{x_i} = \mathsf{receive}(m_i, E) : p_i, \ldots\}\ \{B\}}$$

Notation We will write $\vdash_\Gamma \{A\}\ p\ \{B\}$ to denote that the Hoare triple $\{A\}\ p\ \{B\}$ has a proof in our proof system.

4 Examples

Negociated connection Let us first consider a model of a negociated connection. A server offers a service on a fixed ip address, modeled as a endpoint location – we assume a primitive `bind`(ip) that allocates two endpoints, one of which is allocated at location ip. The server listens for connection requests on the peer of the endpoint ip. No client owns the endpoint ip, but it can try to receive on it; if it succeeds, it gets the right to negociate a connection with the server. The client and server can be modeled as follows

```
server(){
  local f,e';
  [emp]
  f:= bind(ip);
  [$ip \overset{ep}{\mapsto} (C\langle 1\rangle, f) * f \overset{ep}{\mapsto} (\overline{C}\langle 1\rangle, ip)$]
  listen(f);
  [$f \overset{ep}{\mapsto} (\overline{C}\langle 2\rangle, ip)$]
  while(true) {
    e':=accept(f);
    [$f \overset{ep}{\mapsto} (\overline{C}\langle 2\rangle, ip) * e' \overset{ep}{\mapsto} C'\langle q_0\rangle$]
    spawn service(e');
    [$f \overset{ep}{\mapsto} (\overline{C}\langle 2\rangle, ip)$]
  }
}
```

```
client(){
  local f';
  [emp]
  f':=connect(ip);
  [$f' \overset{ep}{\mapsto} \overline{C'}\langle q_0\rangle$]
  run_service(f');
}

// Connection's Contract
contract C {
  initial final state 1
    {?is_listening->2;};
  state 2 {!connect->1;};
}
// Service's contract
contract C' {
  initial state $q_0$  ...
}
```

When the server starts listening, he releases the ownership of ip by sending it towards itself – this is the purpose of the `is_listening` message. To make a connection, a client gets this message, and then send a connection request's message (message `connect`) over ip. For simplicity, we assume that the server accepts any connection request. The connection request thus contains one endpoint allocated by the client to be used by the server to open the service with this client, and it also transfers back the ownership of ip.

```
message is_listening
[$- \overset{ep}{\mapsto} (C\langle 1\rangle, \mathsf{src})$]

message connect(ep)
[$- \overset{ep}{\mapsto} (C\langle 2\rangle, \mathsf{src}) * ep \overset{ep}{\mapsto} C'\langle q_0\rangle$]

listen(f){
  local e';
  [$ip \overset{ep}{\mapsto} (C\langle 1\rangle, f) * f \overset{ep}{\mapsto} (\overline{C}\langle 1\rangle, ip)$]
  send(is_listening,f);
  [$f \overset{ep}{\mapsto} \overline{C}\langle 2\rangle$]
}

accept(f){
  local e';
  [$f \overset{ep}{\mapsto} \overline{C}\langle 2\rangle$]
  e':=receive(connect,f);
  [$f \overset{ep}{\mapsto} (\overline{C}\langle 2\rangle, ip) \quad * \quad ip \overset{ep}{\mapsto} (C\langle 2\rangle, f)$
      $* \quad e' \overset{ep}{\mapsto} C'\langle q_0\rangle$]
  send(is_listening,f);
  return e';
  {$f \overset{ep}{\mapsto} \overline{C}\langle 2\rangle \quad * e' \overset{ep}{\mapsto} C'\langle q_0\rangle$}
}
connect(ip){
  local e',f';
  [emp]
  receive(is_listening,ip);
  [$ip \overset{ep}{\mapsto} C\langle 2\rangle$]
  (e',f') := open(C');
  [$ip \overset{ep}{\mapsto} C\langle 2\rangle \quad * \quad e' \overset{ep}{\mapsto} (C'\langle q_0\rangle, f')$
      $* \quad f' \overset{ep}{\mapsto} (\overline{C}\langle q_0\rangle, e')$]
  send(connect,ip,e');
  [$f' \overset{ep}{\mapsto} \overline{C}\langle q_0\rangle$]
  return f';
}
```

Multicast Consider the following implementation of multicast primitives (limited here to two clients, for simplicity).

```
multicast_send(x,y) [$\mathsf{msnd}(x) * I(y) * I(y)$] {
  send(token,x.0,y,2); receive(ack,x.1)
} [$\mathsf{msnd}(x)$]

multicast_recv(x) [$\mathsf{mrcv}(x)$] {
  local n; (y,n) = receive(token,x.1);
  if n=2 then send(token,x.0,y,1) else send(ack,x.0);
  return y;
} [$\mathsf{mrcv}(x) * I(y)$]
```

To prove this code, $I(.)$ can be any predicate, C must be the contract `contract C {` $2 \xrightarrow{!token} 1 \xrightarrow{!token} 0 \xrightarrow{!ack} 2$ `}`, and

$$\begin{aligned}
\mathsf{mrcv}(x) &\triangleq x \mapsto_{0.25} (-,-)\\
\mathsf{msnd}(x) &\triangleq x \mapsto_{0.5} (X_0, X_1) * X_0 \overset{ep}{\mapsto} (C\langle 2\rangle, X_1) * X_1 \overset{ep}{\mapsto} (\overline{C}\langle 2\rangle, X_0),\\
I_{\text{token}}(y,n) &\triangleq I(y)^n * x \mapsto_{0.5} (X_0, X_1) * X_0 \overset{ep}{\mapsto} (C\langle n-1\rangle, X_1) * X_1 \overset{ep}{\mapsto} (\overline{C}\langle n\rangle, \mathsf{src})\\
I_{\text{ack}} &\triangleq x \mapsto_{0.5} (X_0, X_1) * X_0 \overset{ep}{\mapsto} (C\langle 2\rangle, X_1) * X_1 \overset{ep}{\mapsto} (\overline{C}\langle 0\rangle, \mathsf{src}).
\end{aligned}$$

Let us stress that the I_{token} footprint is implicitly a disjunction on the value of n. The proof of the `receive(token,x.1)` instruction thus invokes first the `receive` rule, then the disjunction rule, and finally the $\text{skip}_{E\lambda,E'}$ rule, which would not have been possible if the rules for `send` and `receive` did not use the pseudo instruction $\text{skip}_{E\lambda,E'}$.

Synchronization barriers Consider the following implementation of a multiple-usage barrier synchronizing N threads. We write F^N to denote $\underbrace{F * \cdots * F}_{N \text{ times}}$.

```
x = new_barrier() [emp] {
  x = new(); (x.0,x.1) = open(C); send(token,x.0,0)
} [barrier(x)]

barrier_wait(x) [$\mathsf{barrier}_{\frac{1}{2N}}(x) * \mathsf{IN}$] {
  local n = receive(token,x.1);
  if n=N-1 then send(ack,x.0,n) else {
    send(token,x.0,n+1);
    n = receive(ack,x.0);
    if n=0 then send(token,x.0,0) else send(ack,x.1,n-1)
  }
} [$\mathsf{barrier}_{\frac{1}{2N}}(x) * \mathsf{OUT}$]

dispose_barrier(x) [$\mathsf{barrier}(x) * \mathsf{OUT}^N$]
  receive(token,x.1); close(x.0,x.1); dispose(x);
} [$\mathsf{OUT}^N$]
```

To prove this code, the formulas IN, OUT in the specification should be any formulas such that $\mathsf{IN}^N \vdash \mathsf{OUT}^N$ and $\mathsf{OUT}^N * \mathsf{IN} \vdash \bot$, and the following contract and auxiliary specifications should be used:

```
contract C { initial state s.0; final state s.1;
  state s.0=r.0 {!token->s.1} state s.N=r.N {?ack->r.(N-1)}
  state s.i (0<i<N) {!token->s.(i+1)}
  state r.i (0<i<n) {?ack->r.(i-1)}}
```

$$\begin{aligned} \mathsf{barrier}_\pi(x) &\triangleq x \mapsto_\pi (-,-) \\ I_{\text{token}}(n) &\triangleq \mathsf{IN}^n * x \mapsto_{0.5} (X_0, X_1) * X_0 \overset{ep}{\mapsto} (C\langle s_{n+1}\rangle, X_1) * X_1 \overset{ep}{\mapsto} (\overline{C}\langle s_n\rangle, \mathsf{src}) \\ I_{\text{ack}}(n) &\triangleq \mathsf{OUT}^n * x \mapsto_{0.5} (X_0, X_1) * X_0 \overset{ep}{\mapsto} (C\langle r_n\rangle, \mathsf{src}) * X_1 \overset{ep}{\mapsto} (\overline{C}\langle r_{n+1}\rangle, X_0) \end{aligned}$$

Internal choice In order to illustrate permission-based sharing and `switch`/`case` rules, consider the following encoding of internal choice

$$\mathit{choice}(p_1, p_2) \triangleq \text{local} \;\; \text{e,f} \;\; \text{in} \;\; \{(\text{e,f})\text{=open}(\text{C});\ \{\text{send}(\text{token,e}) \;||\; p_1' \;||\; p_2'\}\}$$

where $e, f \notin \mathsf{fv}(p_1, p_2)$, and p_i' is defined parametrically in the program p_i by

```
switch { case receive(token,f): {send(notoken,e); $p_i$}
         case receive(notoken,f): {close(e,f)}}
```

It can be checked that $\{A\}$ *choice*(p_1, p_2) $\{B\}$ is derivable from $\{A\}$ p_1 $\{B\}$ and $\{A\}$ p_2 $\{B\}$, giving to each process half of the ownership on the endpoint `f`, and using the following contract and footprint:

contract C { initial state 0 {!token→0; !notoken→1} final state 1;}

$$I_{\text{token}} \triangleq \mathsf{src} \overset{ep}{\hookrightarrow} (C\langle 0\rangle, f) \qquad I_{\text{notoken}} \triangleq \mathsf{src} \overset{ep}{\hookrightarrow} (C\langle 1\rangle, f) * f \underset{0.5}{\overset{ep}{\hookrightarrow}} (C\langle 0\rangle, \mathsf{src})$$

Locks Our last example features an encoding of locks in the heap. The lock is accessed by `x`, and protects a resource $I(.)$ parametrized by `y`:

```
contract C{ initial state 0 {!token->0; !stop->1} final state 1 {}}

new_lock(y;x) [$I(y)$] {
  x=new(); (x.0,x.1) = open(C);
} [$\mathsf{handle}(x) * \mathsf{locked}(x) * I(y)$]

dispose_lock(x,y); [$\mathsf{handle}(x) * \mathsf{locked}(x) * I(y)$] {
  send(stop,x.0); receive(stop,x.1); close(x.0,x.1); dispose(x);
} [$I(y)$]

acquire(x,y) [$\mathsf{handle}_\pi(x)$] {
  receive(token,x.0);
} [$\mathsf{handle}_\pi(x) * \mathsf{locked}(x) * I(y)$]

release(x,y) [$\mathsf{handle}_\pi(x) * \mathsf{locked}(x) * I(y)$] {
  send(token,x.1);
} [$\mathsf{handle}_\pi(x)$]
```

$$\mathsf{locked}(x) \triangleq x \mapsto_{0.5} (X_0, X_1) * X_0 \overset{ep}{\hookrightarrow} (C\langle 0\rangle, X_1) * X_1 \overset{ep}{\hookrightarrow} (\overline{C}\langle 0\rangle, X_0)$$

$$\mathsf{handle}_\pi(x) \triangleq x \mapsto_{\pi/2} (-, -) \qquad I_{\text{token}} \triangleq \mathsf{locked}(\mathsf{x}) * I(y) \qquad I_{\text{stop}} \triangleq \mathsf{emp}$$

Let us stress some subtleties of this encoding and proof:

- We could have derived a proof with permission-based sharing, where endpoints would be part of the handle, and message `token` would transfer $I(y)$ solely; however, this would not ensure that $\mathsf{locked}(x) * \mathsf{locked}(x) \vdash \bot$, which would accept a larger amount of programs with deadlocks.
- The `stop` message is needed to make contract C valid. Without it, we would not be able to guarantee the absence of orphan messages just from the semantics of the CFSM associated to C, and the particular content of message `token` would have to be part of the argument for the absence of orphan messages. Such a `stop` message brings to mind the way channels are sometimes closed one endpoint after the other.

5 Operational Semantics

We now provide an operational semantics to our toy programming language following Villard's thesis [19]. The operational semantics is a binary relation on pairs p, σ, where σ is a *global state* modeling queue contents. We start by defining this object and handy notions related to it.

5.1 Global and Open States

Global states We call *global state* an element $k \in \mathring{\mathsf{EHeap}}$, where

$$\mathring{\mathsf{EHeap}} \triangleq \mathsf{Endpoint} \to \mathsf{Endpoint} \times \mathring{\mathsf{Buffer}} \qquad \mathring{\mathsf{Buffer}} \triangleq (\mathsf{MsgId} \times \mathsf{Val}^* \times \dot{\mathsf{State}})^*$$

When $k(\varepsilon) = (\varepsilon', \mathit{buf})$, we write $\mathit{buffer}(k)(\varepsilon)$ for buf. We define $\mathsf{mate}(k)$ similarly.

Definition 5 (Well-formed global state). *A global state k is well-formed if*

- *for all $\varepsilon \in \mathit{dom}(k)$, $\mathsf{mate}(k)(\mathsf{mate}(k)(\varepsilon)) = \varepsilon \neq \mathsf{mate}(k)(\varepsilon)$;*
- *the set of ε such that $\mathit{buffer}(k)(\varepsilon) \neq \sqcup$ is finite.*

Definition 6 (Environment consistency). *A global state k is consistent with an environment Γ, written $\Gamma \vdash k$, if, for all $\varepsilon, \mathit{buf}, \varepsilon', m, \overrightarrow{v}, \dot{\sigma}$, if $(m, \overrightarrow{v}, \dot{\sigma}) \in \mathit{buf}$ and $k(\varepsilon) = (\varepsilon', \mathit{buf})$ then $\dot{\sigma} \vDash I_m(\varepsilon', \overrightarrow{v})$.*

Open states We call *open state* a pair $\sigma = (\dot{\sigma}, k)$ where $\dot{\sigma}$ is a local state and k a global state.

Definition 7 (Flattable state). *An open state $\sigma = (\dot{\sigma}, k)$ is* flattable *if all local states in*

$$LS_\sigma \triangleq \{\dot{\sigma}\} \cup \{\dot{\sigma}_m : (m, v, \dot{\sigma}_m) \in \mathit{buffer}(k, \varepsilon), \varepsilon \in \mathit{dom}(k)\}$$

are pairwise compatible.

Let $\mathit{emp}(k)$ be the global state obtained by setting all the footprints to $\dot{u}$, *i.e.* by the lifting to k of the function $(a, v, \dot{\sigma}) \mapsto (a, v, \dot{u})$.

Definition 8 (Flattening). *The flattening $\mathit{flat}(\sigma)$ of a flattable open state is defined as*

$$\mathit{flat}(\dot{\sigma}, k) \triangleq \Big(\bigodot_{\dot{\sigma}' \in LS_\sigma} \dot{\sigma}', \mathit{emp}(k) \Big) .$$

Note that $\mathit{flat}(\mathit{flat}(\sigma)) = \mathit{flat}(\sigma)$.

We can now give the definition of well-formed open states, which imposes two restrictions on the flattening of the states additionally to the requirement imposed by the flattening operation itself.

Definition 9 (Well-formed open states). *An open state $\sigma = (\dot{\sigma}, k)$ is well-formed if:*

- *k is well-formed*
- *it is flattable, flat$(\sigma) = ((s_f, h_f, \dot{k}_f), k_f)$, and*
- *$(s_f, h_f, \dot{k}_f)$ is well-formed, and*
- $\mathsf{mate}(k)$ *is an extension of* $\mathsf{mate}(\dot{k}_f)$.

We define a partial composition on well-formed open state: $(\dot{\sigma}_1, k_1)$ and $(\dot{\sigma}_2, k_2)$ are orthogonal if $\dot{\sigma}_1 \perp \dot{\sigma}_2$, $k_1 = k_2 = k$, and $\sigma = (\dot{\sigma}_1 \bullet \dot{\sigma}_2, k)$ is well-formed. In that case, their composition is σ. It can be noticed that well-formed open states equipped with this partial composition form a separation algebra with multiple units [7].

Coherence Between Local and Global States

Definition 10 (Set of configurations of a global state). *Let k be a well-formed global state. To any pair $(\varepsilon, \varepsilon')$ of endpoints such that* $\mathsf{mate}(k)(\varepsilon) = \varepsilon'$*, we associate the set of CFSM configurations* $\mathsf{CONFS}(k, \varepsilon, \varepsilon') \triangleq Q \times Q \times \{w\} \times \{w'\}$*, where $w, w' \in \Sigma^* \times \Sigma^*$ are defined by applying the projection* $(\mathsf{MsgId} \times \mathsf{Val}^* \times \dot{\mathsf{State}}) \to \mathsf{MsgId}$ *to the buffers of ε and ε'.*

Definition 11 (Set of configurations of a local state). *Let $\dot{\sigma}$ be a local state. To any pair of endpoint $(\varepsilon, \varepsilon')$, we associate the set of CFSM configurations*

$$\begin{aligned}&\mathsf{CONFS}(\dot{\sigma}, \varepsilon, \varepsilon')\\ \triangleq&\{(q, q', w, w') : \mathsf{cstate}(\dot{\sigma})(\varepsilon) = q \textit{ if defined, and } \mathsf{cstate}(\dot{\sigma})(\varepsilon') = q' \textit{ if defined}\}\end{aligned}$$

Definition 12 (Set of configurations of an open state). *Let $(\dot{\sigma}, k)$ be a flattable open state, and $flat(\dot{\sigma}, k) = (\dot{\sigma}_f, k_f)$. To any pair of endpoints $(\varepsilon, \varepsilon')$ such that* $\mathsf{mate}(k)(\varepsilon, \varepsilon')$*, we associate the set of CFSM configurations*

$$\mathsf{CONFS}((\dot{\sigma}, k), \varepsilon, \varepsilon') \triangleq \mathsf{CONFS}(\dot{\sigma}_f, \varepsilon, \varepsilon') \cap \mathsf{CONFS}(k, \varepsilon, \varepsilon')$$

We write $\mathsf{CONFS}^{wf}(C)$ for the set of configurations of C that are reachable from the initial configuration.

Definition 13 (Coherence global-local). *A global state k is coherent with a local state $\dot{\sigma}$, written $\Gamma \vdash k \triangleright \dot{\sigma}$, if $(\dot{\sigma}, k)$ is flattable, $flat(\dot{\sigma}, k) = (\dot{\sigma}_f, k_f)$, and for all pair of endpoint $(\varepsilon, \varepsilon')$ such that* $\mathsf{mate}(k)(\varepsilon) = \varepsilon'$ *and* $\mathsf{contract}(\dot{\sigma}_f)(\varepsilon)$ *is defined and equal to C,*

$$\mathsf{CONFS}((\dot{\sigma}, k), \varepsilon, \varepsilon') \cap \mathsf{CONFS}^{wf}(C) \neq \emptyset.$$

5.2 Towards Operational Semantics

For a flattable open state σ, the set of allocated variables and cells (but not the set of allocated endpoints) depends on the portions of state stored inside the buffers, hence on the flattening $\mathit{flat}(\sigma) = ((s_f, h_f, \dot{k}_f), k)$ of σ. We use the following notations:

$$\begin{aligned}
\mathit{valloc}(\sigma) &\triangleq \mathit{dom}(s_f)\\
\mathit{calloc}(\sigma) &\triangleq \mathit{dom}(h_f)\\
\mathit{ealloc}(\sigma) &\triangleq \mathit{dom}(\dot{k}_f) \cup \{\varepsilon : \mathit{buffer}(k)(\varepsilon) \neq \sqcup\}\\
\mathsf{cstate}(\sigma) &\triangleq \mathsf{cstate}(\dot{k}_f)\\
\mathsf{contract}(\sigma) &\triangleq \mathsf{contract}(\dot{k}_f)
\end{aligned}$$

We define shorthands to describe updates in the global state. If $k(\varepsilon) = (\varepsilon', \mathring{\alpha})$, we write $[k \mid \mathit{buffer}(\varepsilon) \leftarrow \mathring{\alpha}']$ for $[k \mid \varepsilon : (\varepsilon', \mathring{\alpha}')]$; if $\dot{k}(\varepsilon) = (\varepsilon', C, r, q)$, we write $\mathsf{cstate}(\dot{k}, \varepsilon)$ for q, $\mathit{role}(\dot{k}, \varepsilon)$ for r, and $[\dot{k} \mid \mathsf{cstate}(\varepsilon) \leftarrow q']$ for $[\dot{k} \mid \varepsilon : (\varepsilon', C, r, q')]$.

We will implicitly consider programs up to a structural congruence relation that treats internal choice $+$, parallel composition $\|$, and case composition as commutative and associative, and sequential composition ; as associative.

The operational semantics implicitly depends on the context Γ, and is of the form

$$p, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p', \sigma' \text{ or } \textbf{error} \ .$$

where σ is assumed in the definition to be flattable (and we will later prove that σ' is flattable as well). The p subscript emphasizes the fact that the transition comes from the program, as the semantics also includes interferences from the environment, of the form

$$\sigma \xrightarrow[\mathsf{e}]{\Gamma} \sigma'$$

to account for messages being sent or received by the other end of a dangling channel (that is, a channel where one end is owned by the program but the other one is not).

The dependence of the semantics in Γ is clearly unrealistic. However, since our semantics will send flattable open states to flattable open states, it is easy to define a semantics that does not depend on Γ, faults less often, and simulates our semantics up to flattening.

Our semantics may raise the following errors:

OwnError indicates an *ownership error*: the program has tried to access a resource it does not currently own, be it a variable, a memory cell or an endpoint;

MsgError indicates a *message error*: either during a reception, an unexpected message is present at the head of a receive buffer, or during closure, one buffer is not empty.

ProtoError indicates that the program is not contract obedient, either because it performs a communication that is not allowed by the contract, or because it closes a channel without having both peers in a final state, or because a `switch`/`case` is not exhaustive.

We write **error** for one of **OwnError**,**MsgError** or **ProtoError** in the rules that propagate an error throughout the programming constructs.

Explicit error detection is a rather unrealistic aspect of the semantics, because threads would not normally fault when merely trying to access the same memory location, and may or may not fault when receiving an unexpected message depending on the implementation. Our semantics can be seen as a "truncation" of a more permissive semantics, for which some states might be reached going through what we consider here as error states. Such a more permissive semantics would still ensure the soundness of the proof system, since proved programs do not go through these error states at any step of their computation.

For conciseness purposes, and whenever possible, we describe all the cases where executing a command will produce an ownership violation together with the reduction where the command executes normally. We do so by putting the premises that are necessary for the command not to fault in $\boxed{\text{boxes}}$. A boxed premise means that there is an additional reduction to **OwnError** from a state where the premise is either false or undefined.

Consider for instance the rule for memory lookup:

$$\frac{\boxed{s(\mathrm{x}) = (1, -)} \qquad \boxed{[\![E]\!]_s} = l \qquad \boxed{h(l)} = (-, v_0, v_1)}{\mathrm{x}\ {=}E.i, ((s, h, \dot{k}), k) \xrightarrow[\mathsf{p}]{\Gamma} {\color{blue}\mathrm{skip}}, (([s \mid \mathrm{x} : (1, v_i)], h, \dot{k}), k)}$$

It indicates that `x =`$E.i$ will fault whenever `x` or one of the variables in E is not present in the current stack with enough permission, or E does not evaluate to an address present in the heap. Thus, this rule stands for the following four rules:

$$\frac{\mathrm{x} \in \mathit{dom}(s) \qquad s(\mathrm{x}) = (1, l) \qquad h(l) = (-, v_0, v_1)}{\mathrm{x}\ {=}E.i, ((s, h, \dot{k}), k) \xrightarrow[\mathsf{p}]{\Gamma} {\color{blue}\mathrm{skip}}, (([s \mid \mathrm{x} : (1, v_i)], h, \dot{k}), k)}$$

$$\frac{\mathrm{x}, \mathit{var}(E) \nsubseteq \mathit{dom}(s)}{\mathrm{x}\ {=}E.i, ((s, h, \dot{k}), k) \xrightarrow[\mathsf{p}]{\Gamma} \textbf{OwnError}} \qquad \frac{s(\mathrm{x}) = (\pi, l) \qquad \pi < 1}{\mathrm{x}\ {=}E.i, ((s, h, \dot{k}), k) \xrightarrow[\mathsf{p}]{\Gamma} \textbf{OwnError}}$$

$$\frac{[\![E]\!]_s = l \qquad l \notin \mathit{dom}(h)}{\mathrm{x}\ {=}E.i, ((s, h, \dot{k}), k) \xrightarrow[\mathsf{p}]{\Gamma} \textbf{OwnError}}$$

5.3 Stack and heap commands.

For the stack and heap commands and constructs, the semantics is derived straightforwardly from the usual semantics, ignoring (almost) the global state.

$$\frac{\boxed{[\![B]\!]_s} = \mathsf{true}}{\mathrm{assume}(B), ((s,h,\dot{k}),k) \xrightarrow[\mathsf{p}]{\Gamma} \mathrm{skip}, ((s,h,\dot{k}),k)}$$

$$\frac{\boxed{s(\mathrm{x}) = (1,-)} \qquad \boxed{[\![E]\!]_s} = v}{\mathrm{x} = E, ((s,h,\dot{k}),k) \xrightarrow[\mathsf{p}]{\Gamma} \mathrm{skip}, (([s \mid \mathrm{x} : (1,v)], h, \dot{k}), k)}$$

$$\frac{\boxed{s(\mathrm{x}) = (1,-)} \qquad l \in \mathsf{Cell} \setminus \mathit{calloc}((s,h,\dot{k}),k)) \qquad v \in \mathsf{Val}}{\mathrm{x} = \mathrm{new}(), ((s,h,\dot{k}),k) \xrightarrow[\mathsf{p}]{\Gamma} \mathrm{skip}, (([s \mid \mathrm{x} : (1,l)], [h \mid l : (1, v_0, v_1)], \dot{k}), k)}$$

$$\frac{\boxed{[\![E]\!]_s} = l \qquad \boxed{h(l) = (1,-,-)}}{\mathrm{dispose}(E), ((s,h,\dot{k}),k) \xrightarrow[\mathsf{p}]{\Gamma} \mathrm{skip}, ((s, h \setminus l, \dot{k}), k)}$$

$$\frac{\boxed{s(\mathrm{x}) = (1,-)} \qquad \boxed{[\![E]\!]_s} = l \qquad \boxed{h(l)} = (-, v_0, v_1)}{\mathrm{x} = E.i, ((s,h,\dot{k}),k) \xrightarrow[\mathsf{p}]{\Gamma} \mathrm{skip}, (([s \mid \mathrm{x} : (1,v_i)], h, \dot{k}), k)}$$

$$\frac{\boxed{h(l) = (1, v_0, v_1)} \qquad \begin{array}{c} \boxed{[\![E_1]\!]_s} = l \qquad \boxed{[\![E_2]\!]_s} = v \\ (i = 0 \wedge v'_0 = v \wedge v'_1 = v_1) \vee (i = 1 \wedge v'_0 = v_0 \wedge v'_1 = v) \end{array}}{E_1.i{=}E_2, ((s,h,\dot{k}),k) \xrightarrow[\mathsf{p}]{\Gamma} \mathrm{skip}, ((s, [h \mid l : (1, v'_0, v'_1)], \dot{k}), k)}$$

Note that the semantics of `new` is the only one that depend on the global state, namely to avoid reusing a location that may be hidden in the contents of a buffer (which could result in a non-flattable state).

5.4 Channel Creation and Destruction

The semantics of open and close takes the protocol of the channel into account: `open` initializes it, and `close` raises a protocol error if the channel is closed in a non-final state of the contract. If the buffers of a closed channel are not empty, it raises a message error. If the endpoints given as arguments to `close` do not form a channel, an ownership error is raised. Like `new`, `open` takes care not to reallocate a location already present in one of the buffers, so as not to create a non-flattable state.

$$\dfrac{\begin{array}{c}\boxed{s(\mathrm{e}) = (1,-)} \qquad \boxed{s(\mathrm{f}) = (1,-)}\\ \varepsilon,\varepsilon' \in \mathsf{Endpoint} \setminus \mathit{ealloc}((s,h,\dot{k}),k) \qquad \mathsf{mate}(k)(\varepsilon) = \varepsilon' \qquad q_0 = \mathsf{init}(C)\\ s' = [s \mid \mathrm{e} : (1,\varepsilon), \mathrm{f} : (1,\varepsilon')] \qquad \dot{k}' = \varepsilon \mapsto (1,\varepsilon',C,(1,q_0)), \varepsilon' \mapsto (1,\varepsilon,\overline{C},(1,q_0))\end{array}}{(\mathrm{e},\mathrm{f})\ = \mathrm{open}(\mathrm{C}), ((s,h,\dot{k}),k) \xrightarrow[\mathsf{p}]{\Gamma} \mathrm{skip}, ((s',h,\dot{k}\cup\dot{k}'),k)}$$

$$\dfrac{\begin{array}{c}\boxed{[\![E_1]\!]_s} = \varepsilon_1\\ \boxed{[\![E_2]\!]_s} = \varepsilon_2 \qquad \boxed{\dot{k}(\varepsilon_1) = (1,\varepsilon_2,-,-)} \qquad \boxed{\dot{k}(\varepsilon_2) = (1,\varepsilon_1,-,-)}\end{array}}{\mathrm{close}\,(E_1,E_2), ((s,h,\dot{k}),k) \xrightarrow[\mathsf{p}]{\Gamma} \mathrm{skip}, ((s,h,\dot{k} \setminus \{\varepsilon_1,\varepsilon_2\}),k)}$$

$$\dfrac{[\![E_1]\!]_s = \varepsilon_1 \qquad [\![E_2]\!]_s = \varepsilon_2 \qquad \mathit{buffer}(k)(\varepsilon_i) \neq \text{\textvisiblespace} \text{ for some } i}{\mathrm{close}\,(E_1,E_2), ((s,h,\dot{k}),k) \xrightarrow[\mathsf{p}]{\Gamma} \mathbf{MsgError}}$$

$$\dfrac{\begin{array}{c}[\![E_1]\!]_s = \varepsilon_1 \qquad [\![E_2]\!]_s = \varepsilon_2 \qquad \dot{k}(\varepsilon_1) = (1,\varepsilon_2,C,(1,q_1))\\ \dot{k}(\varepsilon_2) = (1,\varepsilon_1,C,(1,q_2)) \qquad q_1 \neq q_2 \text{ or } q_1 \notin \mathit{finals}(C)\end{array}}{\mathrm{close}\,(E_1,E_2), ((s,h,\dot{k}),k) \xrightarrow[\mathsf{p}]{\Gamma} \mathbf{ProtoError}}$$

5.5 Sending and Receiving Messages

The semantics of `send` and `receive` is decomposed into an ownership transfer and the update of the endpoint's state. Any of these two steps may fail: the ownership transfer may fail in the send case because a message required by the environment Γ is not available, which raises an ownership error. The update of the endpoint's state may fail, either because the endpoint is not owned with enough permission, which raises an ownership error, or because the contract does not allow the action λ, which raises a protocol error.

$$\frac{\begin{array}{c}\boxed{[\![E]\!]_s} = \varepsilon \qquad \boxed{[\![\overrightarrow{F}]\!]_s} = \overrightarrow{v} \qquad k(\varepsilon) = (\varepsilon', \mathring{\alpha}) \qquad \dot{\sigma}_m \vDash I_m(\varepsilon, \overrightarrow{v}) \\ \textsf{skip}_{\varepsilon!m,\varepsilon'}, (\dot{\sigma}, k) \xrightarrow[\mathsf{p}]{\Gamma} \textsf{skip}, (\dot{\sigma}' \bullet \dot{\sigma}_m, k) \qquad k' = [k \mid \mathit{buffer}(\varepsilon') {\leftarrow} \mathring{\alpha}.(m, \overrightarrow{v}, \dot{\sigma}_m)]\end{array}}{\textsf{send}(\mathrm{m}, E, \overrightarrow{F}), (\dot{\sigma}, k) \xrightarrow[\mathsf{p}]{\Gamma} \textsf{skip}, (\dot{\sigma}', k')}$$

$$\frac{\begin{array}{c}\boxed{[\![E]\!]_s} = \varepsilon \qquad \boxed{[\![\overrightarrow{F}]\!]_s} = \overrightarrow{v} \qquad \mathsf{mate}(k)(\varepsilon) = \varepsilon' \\ \forall.\dot{\sigma}_m \preceq \dot{\sigma}', \dot{\sigma}_m \nvDash I_m(\varepsilon, \overrightarrow{v}) \qquad \textsf{skip}_{\varepsilon!m,\varepsilon'}, (\dot{\sigma}, k) \xrightarrow[\mathsf{p}]{\Gamma} \textsf{skip}, (\dot{\sigma}', k)\end{array}}{\textsf{send}(\mathrm{m}, E, \overrightarrow{F}), (\dot{\sigma}, k) \xrightarrow[\mathsf{p}]{\Gamma} \textbf{OwnError}}$$

$$\frac{\begin{array}{c}\boxed{[\![E]\!]_s} = \varepsilon \qquad \boxed{s(\mathrm{x}_1) = (1, -), \cdots, s(\mathrm{x}_n) = (1, -)} \\ k(\varepsilon) = (\varepsilon', (m, \overrightarrow{v}, \dot{\sigma}_m).\mathring{\alpha}) \qquad s' = [s \mid \overrightarrow{x} : \overrightarrow{(1, v)}] \qquad k' = [k \mid \mathit{buffer}(\varepsilon') {\leftarrow} \mathring{\alpha}]\end{array}}{\overrightarrow{x} {=} \textsf{receive}(\mathrm{m}, E), ((s, h, \dot{k}), k) \xrightarrow[\mathsf{p}]{\Gamma} \textsf{skip}_{\varepsilon?m,\varepsilon'}, ((s', h, \dot{k}) \bullet \dot{\sigma}_m, k')}$$

$$\frac{\begin{array}{c}\boxed{\dot{k}(\varepsilon) = (\pi, \varepsilon', C, (\pi', q))} \\ \boxed{q = q' \text{ or } \pi' = 1} \qquad \mathit{allowed}(C, q, \lambda, q') \qquad \dot{k}' = [\dot{k} \mid \mathsf{cstate}(\varepsilon) {\leftarrow} q']\end{array}}{\textsf{skip}_{\varepsilon\lambda,\varepsilon'}, ((s, h, \dot{k}), k) \xrightarrow[\mathsf{p}]{\Gamma} \textsf{skip}, ((s, h, \dot{k}'), k)}$$

$$\frac{\boxed{\dot{k}(\varepsilon) = (\pi, \varepsilon', C, (\pi', q))} \qquad \nexists q'.\ \mathit{allowed}(C, q, \lambda, q')}{\textsf{skip}_{\varepsilon\lambda,\varepsilon'}, ((s, h, \dot{k}), k) \xrightarrow[\mathsf{p}]{\Gamma} \textbf{ProtoError}}$$

5.6 External Choice

The reduction rules that treat a `switch`/`case` can either succeed and proceed with one of its branches, or fail, either because the receive is prohibited by the protocol, or because an unexpected message is present at the head of one of the inspected buffers, or because, although no unexpected message is necessarily present, the protocol stipulates that a message that is not expected by the program is possibly available.

$$\frac{\overrightarrow{x_i}{=}\mathrm{receive}(m_i, E_i) \xrightarrow[\mathsf{p}]{\Gamma} \mathrm{skip}, \sigma'}{\mathrm{switch}\{\dots, \mathrm{case}\ \overrightarrow{x_i}{=}\mathrm{receive}(m_i, E_i) : \{p_i\}, \dots\}, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p_i, \sigma'}$$

$$\frac{\boxed{[\![E]\!]_s} = \varepsilon \qquad k(\varepsilon) = (-, (m', -, -).\mathring{\alpha}) \qquad (m \neq m' \text{ or } \varepsilon \neq \varepsilon')}{\overrightarrow{x}{=}\mathrm{receive}(m, E), (\dot{\sigma}, k) \xrightarrow[\mathsf{p}]{\Gamma} \mathsf{WARNING}(\varepsilon')}$$

$$\frac{\overrightarrow{x_i}{=}\mathrm{receive}(m_i, E_i) \xrightarrow[\mathsf{p}]{\Gamma} \mathsf{WARNING}([\![\mathsf{E_j}]\!]_\mathsf{s}) \text{ for one } j \text{ and for all } i}{\mathrm{switch}\{\dots, \mathrm{case}\overrightarrow{x_i}{=}\mathrm{receive}(m_i, E_i) : \{p_i\}, \dots\}, \sigma \xrightarrow[\mathsf{p}]{\Gamma} \mathbf{MsgError}}$$

$$\frac{\begin{array}{c}\boxed{[\![E_i]\!]_s = \varepsilon} \text{ for all i} \\ \boxed{\dot{k}(\varepsilon) = (-, C, -, (-, q))} \qquad \mathsf{choices}(C, q) \not\subseteq \{m_1, \dots, m_n\}\end{array}}{\mathrm{switch}\{\dots, \mathrm{case}\overrightarrow{x_i}{=}\mathrm{receive}(m_i, E_i) : \{p_i\}, \dots\}, \sigma \xrightarrow[\mathsf{p}]{\Gamma} \mathbf{ProtoError}}$$

5.7 Programming constructs.

We introduce the predicate $r\mathring{a}ce(p_1, p_2, \sigma)$, true if it is impossible to partition σ into two disjoint substates on which each program can safely make a step.

Definition 14 (Race Detection). *$r\mathring{a}ce(p_1, p_2, \sigma)$ holds if and only if for all well-formed open states σ_1 and σ_2 such that $\sigma_1 \bullet \sigma_2 = \sigma$, either $p_1, \sigma_1 \xrightarrow[\mathsf{p}]{\Gamma}$* **OwnError***, or $p_2, \sigma_2 \xrightarrow[\mathsf{p}]{\Gamma}$* **OwnError***.*

This predicate is used to generate ownership error in a special rule handling parallel composition, which is otherwise described by a standard interleaving semantics. The semantics of the remaining programming constructs is standard.

$$p_1 + p_2, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p_1, \sigma \qquad\qquad p^*, \sigma \xrightarrow[\mathsf{p}]{\Gamma} \mathrm{skip} + (p;\ p^*), \sigma$$

$$\frac{p_1, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p_1', \sigma'}{p_1;\ p_2, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p_1';\ p_2, \sigma'} \qquad\qquad \mathrm{skip};\ p_2, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p_2, \sigma \qquad\qquad \frac{p_1, \sigma \xrightarrow[\mathsf{p}]{\Gamma} \mathbf{error}}{p_1;\ p_2, \sigma \xrightarrow[\mathsf{p}]{\Gamma} \mathbf{error}}$$

$$\frac{\mathit{r\mathring{a}ce}(p_1, p_2, \sigma)}{p_1 \parallel p_2, \sigma \xrightarrow[\mathsf{p}]{\Gamma} \mathbf{OwnError}} \qquad\qquad \frac{p_1, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p_1', \sigma'}{p_1 \parallel p_2, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p_1' \parallel p_2, \sigma'} \qquad\qquad \frac{p_1, \sigma \xrightarrow[\mathsf{p}]{\Gamma} \mathbf{error}}{p_1 \parallel p_2, \sigma \xrightarrow[\mathsf{p}]{\Gamma} \mathbf{error}}$$

$$\frac{v \in \mathsf{Val} \qquad \mathrm{y} \notin \mathit{valloc}(\sigma) \cup \mathit{freevar}(p)}{\mathrm{local}\ \ \mathrm{x}\ \ \mathrm{in}\ p, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p[\mathrm{x}\leftarrow\mathrm{y}];\ \ \mathrm{delete}\,(\mathrm{y}), \sigma[s\leftarrow s \uplus \{y \mapsto v\}]}$$

$$\frac{\mathrm{y} \in \mathit{dom}(s)}{\mathrm{delete}\,(\mathrm{y}), ((s, h, \dot{k}), k) \xrightarrow[\mathsf{p}]{\Gamma} \mathrm{skip}, ((s \setminus \{\mathrm{y}\}, h, \dot{k}), k)}$$

Note also that, in the treatment of the local variable construct, as in the case of `new`, we now have to take into account not only the domain of the surface stack, but also all the variables that are allocated in the footprints of the messages of all buffers, in order to stay away from ill-formed states.

5.8 Interferences

Interferences from the environment are described by a single rule, given below. The rule transforms an open state into an equivalent one with respect to its local state, but the contents of the buffers may have changed. These changes include the possibility for the environment to perform sends and receives on endpoints that are not directly controlled by the program, in accordance to their contracts, and to open and close channels not visible to the program.

$$\frac{(\dot{\sigma}, k') \text{ is flattable} \qquad \Gamma \vdash k' \qquad \Gamma \vdash k' \triangleright \dot{\sigma}}{(\dot{\sigma}, k) \xrightarrow[\mathsf{e}]{\Gamma} (\dot{\sigma}, k')} \qquad\qquad \frac{\sigma \xrightarrow[\mathsf{e}]{\Gamma} \sigma'}{p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma} p, \sigma'} \qquad\qquad \frac{p, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p', \sigma'}{p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma} p', \sigma'}$$

Note that these interferences are actually a permissive over-approximation of what a real environment might do: with this definition, the environment may also modify the buffers of endpoints owned by the program, and even of endpoints in the local state of the program, provided that it leaves the buffers in a state coherent with what the protocols of these endpoints state. This is the simplest way we found to formalize an environment.

The fact that it really over-approximates an environment will be clarified by the parallel decomposition lemma (see Lemma 8 below).

6 Soundness

In this section, we establish the soundness of the proof system with respect to the operational semantics we just defined. The proof goes in two steps: first, we collect some important properties of the operational semantics that are standard for proofs of soundness of Separation Logic (locality, parallel decomposition), and another property that are reminiscent of session types (subject reduction). We then put together these properties to formally prove that all derivable Hoare triples are valid.

6.1 Structural Properties of the Operational Semantics

Lemma 1. *For all p, p', σ, σ', if $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma} p', \sigma'$, and σ is flattable, then σ' is flattable.*

Proof. We mentioned that the only cases that required a special attention while designing the semantics were the allocations, that we carefully considered "fresh" with the flattening of σ, and not just the local state. □

Lemma 2. *For all p, p', σ, σ', if $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma} p', \sigma'$, and σ is well-formed, then σ' is well-formed.*

Proof. Straightforward. □

Lemma 3. *For all $p, p', \dot{\sigma}, \dot{\sigma}', k, k'$, if $p, (\dot{\sigma}, k) \xrightarrow[\mathsf{p,e}]{\Gamma} p', (\dot{\sigma}', k')$ and $\Gamma \vdash k$, then $\Gamma \vdash k'$.*

Proof. The only non-trivial case is the send case, where it is required to check that the local state added in k satisfies its footprint. For other cases, the set of local states stored in k may only decrease, hence the result. □

For a contract C, let $\rightarrow_C$ be the relation over sets of configurations of the CFSM associated to C defined by lifting the reachability relation $\rightarrow$ to sets of states as follows:

$$S \rightarrow_C S' \triangleq \forall \gamma' \in S'. \exists \gamma \in S. \gamma \rightarrow \gamma' \ .$$

We write $\rightarrow_C^{=}$ to denote the reflexive closure of this relation.

Lemma 4. *For all $p, p', \sigma, \sigma', \varepsilon, \varepsilon'$, if*

- $p, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p', \sigma'$

- $\mathsf{mate}(\sigma')(\varepsilon) = \varepsilon'$
- $\mathsf{contract}(\sigma')(\varepsilon) = C$

then

1. *either* $\mathsf{contract}(\sigma)(\varepsilon)$ *is undefined, and* $\mathsf{CONFS}(\sigma', \varepsilon, \varepsilon') = \{(q_0, q_0, \sqcup, \sqcup)\}$
2. *or* $\mathsf{mate}(\sigma)(\varepsilon) = \varepsilon'$*,* $\mathsf{contract}(\sigma)(\varepsilon) = C$ *and*

$$\mathsf{CONFS}(\sigma, \varepsilon, \varepsilon') \to_C^{=} \mathsf{CONFS}(\sigma, \varepsilon, \varepsilon')\ .$$

Note that this lemma would not hold if $p, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p', \sigma'$ were replaced by $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma} p', \sigma'$ in the hypothesis.

Proof. If the transition $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma} p', \sigma'$ is not a channel operation, $k = k'$, and the result is immediate. Let us analyze the remaining cases. Let us fix some ε satisfying all the above hypothesis.

Let us assume first that ε is not one of the two endpoints of the channel over which the instruction applies. It can be observed that $\mathsf{cstate}(\varepsilon)$ is either undefined in both σ and σ', or defined in both σ and σ', and in that case $\mathsf{cstate}(\sigma)(\varepsilon) = \mathsf{cstate}(\sigma')(\varepsilon)$ (recall that $\mathsf{cstate}(\sigma)$ is defined according to the flattening). Thus, for an endpoint that is not one of the endpoints of the channel over which some action is performed, the constraint is the same in σ and σ', and $\mathsf{CONFS}(\sigma, \varepsilon, \varepsilon') = \mathsf{CONFS}(\sigma', \varepsilon, \varepsilon')$, which shows that case 2 above holds.

Let us assume now that ε is one of the two endpoints of the channel over which the instruction applies, and reason by case analysis on the instruction.

- `close` instruction: Since $\mathsf{contract}(\sigma')(\varepsilon) = C$ is defined by hypothesis, ε is not the closed channel. This case is thus impossible.
- `open` instruction: case 1 above holds.
- `send` instruction: let us show that the case 2 above holds, and in particular that $\mathsf{CONFS}(\sigma, \varepsilon, \varepsilon') {\to} \mathsf{CONFS}(\sigma', \varepsilon, \varepsilon')$. By symmetry, we may assume that ε is the endpoint used for sending. By the rules of `send` and $\mathrm{skip}_{\varepsilon\lambda,\varepsilon'}$, $\mathsf{cstate}(\sigma, \varepsilon)$ and $\mathsf{cstate}(\sigma', \varepsilon)$ must be defined, say respectively q_1 and q_2. Moreover, these rules also ensure that there is a transition $q \xrightarrow{\lambda} q'$ in C. Let $\gamma_2 = (q_2, q_2', w_2, w_2')$ be a configuration in $\mathsf{CONFS}(\sigma', \varepsilon, \varepsilon')$. Then $w_2' = w_1'.m$, and $\gamma_1 = (q, q_2', w_2, w_1') \to \gamma_2$. Moreover, $\gamma_1 \in \mathsf{CONFS}(\sigma', \varepsilon, \varepsilon')$ since, again, q_2' is constrained in the same way in σ and σ'.
- `receive` instruction: same arguments as for `send`. □

Lemma 5 (Subject Reduction). *If* $p, (\dot\sigma, k) \xrightarrow[\mathsf{p,e}]{\Gamma} p', (\dot\sigma', k')$ *and* $\Gamma \vdash k \triangleright \dot\sigma$*, then* $\Gamma \vdash k' \triangleright \dot\sigma'$*.*

Proof. If $p, (\dot\sigma, k) \xrightarrow[\mathsf{p,e}]{\Gamma} p', (\dot\sigma', k')$ by an interference, then the lemma is true by definition of interferences. Otherwise, let us apply the definition of $\Gamma \vdash k \triangleright \dot\sigma$. First $(\dot\sigma', k')$ is flattable by Lemma 1. Second, let $(\varepsilon, \varepsilon')$ be a pair of endpoint such that $\mathsf{mate}(k)(\varepsilon) = \varepsilon'$ and $\mathsf{contract}(\dot{\sigma}_f)(\varepsilon)$ is defined and equal to C. We need to show that

$$\mathsf{CONFS}((\dot{\sigma}', k'), \varepsilon, \varepsilon') \cap \mathsf{CONFS}^{wf}(C) \neq \emptyset \ .$$

We may apply the previous lemma. If we are in the first case, the result is immediate. If we are in the second case, then

$$\mathsf{CONFS}((\dot{\sigma}, k), \varepsilon, \varepsilon') \to_C^{=} \mathsf{CONFS}((\dot{\sigma}', k'), \varepsilon, \varepsilon'),$$

and from the hypothesis $\Gamma \vdash k \rhd \dot{\sigma}$, $\mathsf{CONFS}((\dot{\sigma}, k), \varepsilon, \varepsilon') \cap \mathsf{CONFS}^{wf}(C) \neq \emptyset$, which ends the proof. □

Lemma 6. *For all $\dot{\sigma}_1, \dot{\sigma}_2, k$, if $\Gamma \vdash k \rhd \dot{\sigma}_1 \bullet \dot{\sigma}_2$, then $\Gamma \vdash k \rhd \dot{\sigma}_1$.*

Proof. Let us fix some ε such that $\mathsf{contract}(\dot{\sigma}_1, k)(\varepsilon) = C$ is defined, and let $\varepsilon' = \mathsf{mate}(k)(\varepsilon)$. By hypothesis and definition of the global-local coherence,

$$\mathsf{CONFS}(\dot{\sigma}_1 \bullet \dot{\sigma}_2, k, \varepsilon, \varepsilon') \cap \mathsf{CONFS}^{wf}(C) \neq \emptyset \ .$$

By definition of $\mathsf{CONFS}(\dot{\sigma}, k, \varepsilon, \varepsilon')$,

$$\mathsf{CONFS}(\dot{\sigma}_1 \bullet \dot{\sigma}_2, k, \varepsilon, \varepsilon') \subseteq \mathsf{CONFS}(\dot{\sigma}_1, k, \varepsilon, \varepsilon') \ .$$

Thus $\mathsf{CONFS}(\dot{\sigma}_1, k, \varepsilon, \varepsilon') \cap \mathsf{CONFS}^{wf}(C) \neq \emptyset$, which ends the proof. □

In the two lemmas below, we write **error** to denote either **OwnError** or **ProtoError**.

Lemma 7 (Locality). *For all program p and open states σ_1 and σ_2 such that $\sigma_1 \bullet \sigma_2 = (\dot{\sigma}, k)$ is defined and $\Gamma \vdash k \rhd \dot{\sigma}$,*

1. *if $p, \sigma_1 \bullet \sigma_2 \xrightarrow[\mathsf{p,e}]{\Gamma}$ **error** then $p, \sigma_1 \xrightarrow[\mathsf{p,e}]{\Gamma}$ **error**.*
2. *if $p, \sigma_1 \bullet \sigma_2 \xrightarrow[\mathsf{p,e}]{\Gamma} p', \sigma'$ then either $p, \sigma_1 \xrightarrow[\mathsf{p,e}]{\Gamma}$ **error** or there exists σ_1', σ_2' such that*
   - *$\sigma' = \sigma_1' \bullet \sigma_2'$;*
   - *$p, \sigma_1 \xrightarrow[\mathsf{p,e}]{\Gamma} p', \sigma_1'$;*
   - *$\sigma_2 \xrightarrow[\mathsf{e}]{\Gamma} \sigma_2'$.*

Proof. The first part of the lemma, sometimes called “safety monotonicity”, is straightforward by induction on the derivation tree of $p, \sigma_1 \bullet \sigma_2 \xrightarrow[\mathsf{e}]{\Gamma}$ **error**. Let us prove the second part. Assume $\sigma_1 = (\dot{\sigma}_1, k_1)$, $\sigma_2 = (\dot{\sigma}_2, k_2)$ and $\sigma' = (\dot{\sigma}', k')$ are as above, and let us reason by induction on the derivation tree of $p, \sigma_1 \bullet \sigma_2 \xrightarrow[\mathsf{p,e}]{\Gamma} p', \sigma'$.

Assume first that $p, \sigma_1 \bullet \sigma_2 \xrightarrow[\mathsf{p}]{\Gamma} p', \sigma'$, and that $p, \sigma_1 \not\xrightarrow[\mathsf{p,e}]{\Gamma}$ **error**. The key observation is that changes between $\dot{\sigma}_1 \bullet \dot{\sigma}_2$ and $\dot{\sigma}'$ concern only the resources that are requested for avoiding an ownership error. The only subtlety is in the `send` case, where this observation would be false if we did not have precise footprint formulas. From this observation, we have $\dot{\sigma}' = \dot{\sigma}'_1 \bullet \dot{\sigma}_2$ for some $\dot{\sigma}'_1 \preceq \dot{\sigma}'$ such that $p, \sigma_1 \xrightarrow[\mathsf{p,e}]{\Gamma} p', (\dot{\sigma}'_1, k')$ Since $\Gamma \vdash k \triangleright \dot{\sigma}_1 \bullet \dot{\sigma}_2$, $\Gamma \vdash k' \triangleright \dot{\sigma}'_1 \bullet \dot{\sigma}_2$ by subject reduction, and thus $\Gamma \vdash k' \triangleright \dot{\sigma}_2$ by Lemma 6. Then $\sigma_2 \xrightarrow[\mathsf{e}]{\Gamma} \sigma'_2$ for $\sigma'_2 = (\dot{\sigma}_2, k')$, which ends the proof.

Assume now that $p, \sigma_1 \bullet \sigma_2 \xrightarrow[\mathsf{e}]{\Gamma} p', \sigma'$. Then $\sigma' = (\dot{\sigma}_1, \dot{\sigma}_2, k')$, and $\Gamma \vdash k' \triangleright \dot{\sigma}'_1 \bullet \dot{\sigma}_2$. Choosing $\sigma'_i = (\dot{\sigma}_i, k')$ ends the proof by Lemma 6. □

Lemma 8 (Parallel decomposition). *For all pair of programs p_1, p_2, for all state σ and for all σ_1, σ_2 such that $\sigma_1 \bullet \sigma_2 = \sigma$,*

1. *if $p_1 \parallel p_2, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma}$ **error** then $p_1, \sigma_1 \xrightarrow[\mathsf{p,e}]{\Gamma}$ **error** or $p_2, \sigma_2 \xrightarrow[\mathsf{p,e}]{\Gamma}$ **error**;*
2. *if $p_1 \parallel p_2, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma} p'_1 \parallel p'_2, \sigma'$ then $p_1, \sigma_1 \xrightarrow[\mathsf{p,e}]{\Gamma}$ **error** or $p_2, \sigma_2 \xrightarrow[\mathsf{p,e}]{\Gamma}$ **error** or there are orthogonal states σ'_1, σ'_2 such that $\sigma' = \sigma'_1 \bullet \sigma'_2$ and*
   - $p_1, \sigma_1 \xrightarrow[\mathsf{p,e}]{\Gamma} p'_1, \sigma'_1$
   - $p_2, \sigma_2 \xrightarrow[\mathsf{p,e}]{\Gamma} p'_2, \sigma'_2$.

Proof. The first point is a direct consequence of the rules for error propagation and the locality lemma. Let us prove the second case. Assume p_1, p_2, σ_1, σ_2 as above, and assume moreover that $p_1, \sigma_1 \not\xrightarrow[\mathsf{p,e}]{\Gamma}$ **error** and $p_2, \sigma_2 \not\xrightarrow[\mathsf{p,e}]{\Gamma}$ **error**. Let us reason by case analysis on the first rule applied in the derivation tree of $p_1 \parallel p_2, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma} p'_1 \parallel p'_2, \sigma'$. There are only two possible cases:

- the first rule applied is the interleaving rule

$$\frac{p_1, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p'_1, \sigma'}{p_1 \parallel p_2, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p'_1 \parallel p_2, \sigma'}\ .$$

  Then by locality lemma, there are σ'_1, σ'_2 such that $\sigma' = \sigma'_1 \bullet \sigma'_2$, $p_1, \sigma_1 \xrightarrow[\mathsf{p}]{\Gamma} p'_1, \sigma'_1$, and skip$, \sigma_2 \xrightarrow[\mathsf{e}]{\Gamma}$ skip$, \sigma'_2$. We are thus left to prove $p_2, \sigma_2 \xrightarrow[\mathsf{p,e}]{\Gamma} p'_2, \sigma'_2$, which is a direct consequence of skip$, \sigma_2 \xrightarrow[\mathsf{e}]{\Gamma}$ skip$, \sigma'_2$.
- the first rule applied is the interference rule. Setting $\sigma_1 = (\dot{\sigma}_1, k)$, $\sigma_2 = (\dot{\sigma}_2, k)$, and $\sigma' = (\dot{\sigma}', k')$, we have thus $\dot{\sigma}' = \dot{\sigma}_1 \bullet \dot{\sigma}_2$ and $\Gamma \vdash k' \triangleright \dot{\sigma}'$, so $\Gamma \vdash k' \triangleright \dot{\sigma}_1$ and $\Gamma \vdash k' \triangleright \dot{\sigma}_2$ by Lemma 6, which ends the proof choosing $\sigma'_i = (\dot{\sigma}_i, k')$. □

6.2 Partial Correctness

Definition 15 (Validity). *A triple is* valid with respect to a footprint context *Γ, written $\vDash_\Gamma \{A\}\ p\ \{B\}$, if, for all instantiation of the free logical variables of A and B, for all state $\sigma = (\dot\sigma, k)$, if the following holds:*

- $\Gamma \vdash k \triangleright \dot\sigma$
- $\Gamma \vdash k$
- $\dot\sigma \vDash A$

then the following properties hold:

1. $p, \sigma \not\xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*}$ **OwnError***;*
2. $p, \sigma \not\xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*}$ **ProtoError***;*
3. *if* $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*}$ skip, $(\dot\sigma', k')$*, then* $\dot\sigma' \vDash B$*.*

Theorem 1 (Soundness). *If* $\vdash_\Gamma \{A\}\ p\ \{B\}$ *then* $\vDash_\Gamma \{A\}\ p\ \{B\}$*.*

Proof. By induction on the derivation tree of $\{A\}\ p\ \{B\}$. The case of small axioms and structural rules is by definition of $\xrightarrow[\mathsf{p,e}]{\Gamma}$. The case of the frame and parallel rules are respectively handled by the locality and parallel decomposition lemmas. □

7 Proved Programs Are Safe

7.1 Defining Safety and Provable Safety

Let us briefly recall the safety properties we aim at.

Absence of memory violations no dangling pointer is dereferenced, modified, or disposed.
Absence of races two threads never simultaneously try to access a variable or a memory location when at least one of them is a write access.
Absence of memory leaks the program never runs into a state preventing all continuations from deallocating a channel or a cell.
Absence of orphan messages buffers are always empty when they are closed.
Absence of unspecified receptions all switch receive constructs are exhaustive.

Our operational semantics features some error states that model these errors (except memory leaks). Theo. 1 thus already ensures that provable programs do not reach memory violations or races according to the operational semantics we defined. This result is however a bit limited, because:

- it is based on a semantics that depend on the proof environment Γ, which is an arguably undesirable form of circularity;

- it does not say anything about memory leaks, orphan messages, and unspecified receptions.

We introduce now an operational semantics that does not depend on the annotations. Γ. For a proof environment Γ, we write $\mathsf{emp}(\Gamma)$ for the proof environment in which:

- all message footprints are replaced by emp,
- all contracts are replaced by a single state contract that allows any send and receive from this state.

We model run-time executions by the ones of the operational semantics with empty footprints and universal contracts, without interferences from the environment[6], and without checks on the contract obedience.

Definition 16 (Run-time semantics). *We say that p, σ evolves to p', σ' (resp.* **error***) at run-time, $p, \sigma \Rightarrow p', \sigma'$, if*

- *either $p, \sigma \xrightarrow[\mathsf{p}]{\mathsf{emp}(\Gamma)} p', \sigma'$ for any Γ;*
- *or $p, \sigma \xrightarrow[\mathsf{p}]{\mathsf{emp}(\Gamma)}$* **error** *and* **error** $\neq$ **ProtoError***.*

Defining the notion of safety we aim at prompts us to clarify the notion of memory leaks, which has not been properly modeled so far. We assume a fixed set G of global variables, and we write emp_G to denote the open state σ such that $calloc(\sigma) = ealloc(\sigma) = \emptyset$, and $valloc(\sigma) = G$, *i.e.* the state in which all cells and endpoints have been deallocated.

Definition 17 (Leak-free state). *An open state σ is said* leak-free *if there is a program p such that $p, \sigma \Rightarrow^* \mathsf{emp}_G$.*

Definition 18 (Safety). *A program p is said safe if*

1. *$p, \mathsf{emp}_G \not\Rightarrow^*$* **error***, and*
2. *if $p, \mathsf{emp}_G \Rightarrow^* \sigma$, then σ is leak-free.*

Let us now clarify how the proof system should be used to prove safety. First, one has to be careful to use contracts that prevent unspecified receptions and message leaks. This is however not enough: even if message leaks are forbidden by contracts, some memory leaks may not be visible in the proof. For instance, the Hoare triple

$$\{e, f \Vdash \mathsf{emp}\}\ (\mathrm{e,f})\ = \mathrm{open}(\mathrm{C});\ \mathrm{send}(\mathrm{channel,e,e})\ \{e, f \Vdash \mathsf{emp}\}$$

is derivable in the proof environment annotating the `channel` message with the footprint $\mathsf{src} \overset{ep}{\mapsto} (C\langle - \rangle, X) * X \overset{ep}{\mapsto} (\overline{C}\langle - \rangle, \mathsf{src})$, although this program did not deallocate all the memory. We thus need to restrict environments in a way that

[6] Avoiding interferences is a simplification. We could actually allow interferences, provided they do not introduce memory leaks, but avoid such a complication for clarity reasons.

forbids such situations. The precise definition of valid environment will be later detailed (see Def. 21), let us just now give a blackboxed definition of provable safety. We say that a program p terminates[7] if for all σ, there is σ', such that either $p, \sigma \Rightarrow^* \text{skip}, \sigma'$ or $p, \sigma \Rightarrow^* \textbf{error}$.

Definition 19 (Provable Safety). *A program p is said* provable safe *if there is a program p', and an admissible environment Γ such that* $\vdash_\Gamma \{G \Vdash \mathsf{emp}\}\ p; p'\ \{G \Vdash \mathsf{emp}\}$ *and p' is terminates.*

7.2 Runtime Soundness

Proving that provable safe programs are safe requires quite a lot of work. First, we need to link **ProtoError** to **MsgError** using contracts. Second, we need to link the runtime semantics to the proof-dependent operational semantics. Third, we need to link the absence of memory leaks in the proof to the same property for the runtime semantics.

The first point proceeds directly from the subject reduction lemma.

Lemma 9 (Message Safety). *For all program p, for all state $\sigma = (\dot{\sigma}, k)$, if the following holds*

- $\Gamma \vdash k \triangleright \dot{\sigma}$
- $p, \sigma \not\xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*}$ **ProtoError**
- $p, \sigma \not\xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*}$ **OwnError**

then $p, \sigma \not\xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*}$ **MsgError**.

Proof. Let us assume the hypothesis.

It can be observed that the semantics is designed in such a way that $p, \sigma \xrightarrow[\mathsf{e}]{\Gamma}\xrightarrow[\mathsf{p}]{\Gamma}$ **MsgError** occurs due to an unspecified reception if and only if there are $\varepsilon, \varepsilon'$ such that the set of configurations $\mathsf{CONFS}(\sigma, \varepsilon, \varepsilon')$ contains a configuration in an unspecified reception Similarly, $p, \sigma \xrightarrow[\mathsf{e}]{\Gamma}\xrightarrow[\mathsf{p}]{\Gamma}$ **MsgError** occurs due to an orphan message if and only if there are $\varepsilon, \varepsilon'$ such that the set of configurations $\mathsf{CONFS}(\sigma, \varepsilon, \varepsilon')$ contains a final non-stable configuration and p may close the channel $\varepsilon, \varepsilon'$.

Assume, by contradiction, that $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*}$ **MsgError**, and let p', and $\sigma' = (\dot{\sigma}', k')$ be such that:

[7] We may wonder wether the assumption that p' terminates in Def 19 would be easy to prove or not. On the one hand, if p' is sequential, and deadlock-free, it is sufficient to prove that from any program point, a dispose instruction is always reachable, which is quite likely to be the case for the p' one may think about for standard data structures. However, if p' intends to model a distributed garbage collector, it is likely that a lot of work would have to be done for proving this assumption.

- $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*} p', \sigma'$
- $p', \sigma' \xrightarrow[\mathsf{e}]{\Gamma}\xrightarrow[\mathsf{p}]{\Gamma}$ **MsgError**,
- for all p'', σ'' such that $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*} p'', \sigma'' \xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*} p', \sigma'$, $p'', \sigma'' \not\xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*}$ **MsgError**.

By the previous observation, there is some faulty configuration (q, q', w, w') in $\mathsf{CONFS}(\sigma', \varepsilon, \varepsilon')$. By subject reduction, $\Gamma \vdash k' \triangleright \dot{k}'$, and thus $\mathsf{CONFS}(\sigma', \varepsilon, \varepsilon')$ contains also a reachable configuration (q_1, q'_1, w, w'). Since the contract is valid, these two configurations must be distinct. Let us show that they can be taken equal:

- if the error is an orphan message, then $\varepsilon, \varepsilon'$ must be owned at closure, so their states are constraint, and $\mathsf{CONFS}(\sigma', \varepsilon, \varepsilon')$ contains only one configuration.
- if the error is an unspecified reception, then ε must be owned, but note that $\mathsf{cstate}(\sigma')(\varepsilon')$ may be undefined. This means $q_1 = q$ but not necessarily $q'_1 = q'$. However, due to the definition of unspecified receptions, (q, q'_1, w, w') is also an unspecified reception, hence the contradiction. □

The second step, *i.e.* the connection between the runtime semantics and the proof-based semantics, goes through the operation of flattening we introduced in previous sections (Def 8). For an open state σ, we write $\mathit{erase}(\sigma)$ to denote the open state obtained from $\mathit{flat}(\sigma)$ by replacing all contracts by the universal single state contract.

Lemma 10 (Runtime Soundness). *For all p, σ, σ',*

1. *if $p, \mathit{erase}(\sigma) \Rightarrow$ **error**, then $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma}$ **error**;*
2. *if $p, \mathit{erase}(\sigma) \Rightarrow p', \sigma'$, then*
   - *either $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma}$ **error**;*
   - *or $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*} p', \sigma''$, for some σ'' such that $\mathit{erase}(\sigma'') = \sigma'$.*

By straightforward induction, this lemma states that any error in the runtime semantics can be lifted to an error in the proof-dependent semantics: if $p, \sigma \Rightarrow^{*}$ **error**, then $p, \sigma \xrightarrow[\mathsf{p,e}]{\Gamma}{}^{*}$ **error**.

Proof. Assume $\mathit{erase}(\sigma) = (\dot{\sigma}, \dot{k})$ and $\sigma = (\dot{\sigma}_1, \dot{k}_1)$. Let us first prove the first point of Lemma 10. Assume $p, \mathit{erase}(\sigma) \Rightarrow$ **error**; one of the following cases holds:

- a **OwnError** is triggered: then it is also triggered by $\xrightarrow[\mathsf{p}]{\Gamma}$ thanks to safety monotonicity (observe that $\dot{\sigma}_1 \preceq \dot{\sigma}$);
- a **MsgError** is triggered: the error depends only on the first message identifier in the queue causing the error, which is the same in σ and $\mathit{erase}(\sigma)$.
- a **ProtoError** is triggered: this is never the case, due to Def 16.

Let us now prove the second point of Lemma 10. Assume $p, \mathit{erase}(\sigma) \Rightarrow p', \sigma'$; one of the following cases holds:

- a cell-manipulating instruction is executed: then the result holds by locality and the fact that $\dot{\sigma}_1 \preceq \dot{\sigma}$.
- a channel instruction is executed: if it does not fail for $\xrightarrow[\mathsf{p}]{\Gamma}$, it possibly differs from $\Rightarrow$ by transferring a message footprint, thus $\mathit{erase}(\sigma'') = \sigma'$.

□

7.3 Tracking Memory Leaks

The third step towards our main result on safety is to show that the logic correctly tracks memory leaks. As already observed, this is not true for every environment proof Γ. Intuitively, the problem comes from the following gap between runtime and symbolic execution:

- at runtime, the open state always remain self-contained: all endpoints that have been allocated remain allocated until their closure;
- symbolic execution, on the contrary, allows the state to appear not self-contained

Example 6. In the example triple

$$\{e, f \Vdash \mathsf{emp}\}\ (\mathrm{e},\mathrm{f})\ = \mathrm{open}(\mathrm{C});\ \mathrm{send}(\mathrm{channel},\mathrm{e},\mathrm{e})\ \{e, f \Vdash \mathsf{emp}\}$$

the endpoints e and f become ownerless in the final state of the program. The notion of ownership we care about is an indirect, recursive one, where the owner is a thread; for instance, in the following example

$$\begin{gathered}\{e_1, e_2, f_1, f_2 \Vdash \mathsf{emp}\}\\ (e_1, f_1) = \mathrm{open}(C); (e_2, f_2) = \mathrm{open}(C); \mathrm{send}(ep, e_2, e_2); \mathrm{send}(ep, e_1, f_2)\\ \{e_1, e_2, f_1, f_2 \Vdash e_1 \overset{ep}{\mapsto} (C\langle-\rangle, f_1) * f_1 \overset{ep}{\mapsto} (\overline{C}\langle-\rangle, e_1)\}\end{gathered}$$

the endpoints e_2, f_2 are not ownerless: they are indirectly owned by the thread through f_1. If the continuation `send`(ep, e_1, f_1) is considered, however, f_1, e_2, f_2 become ownerless.

We will need to forbid ownerless cells to precisely track garbage cells, but this is due to a technical limitation, and we should make clear that ownerless cells and garbage cells are different. A cell can be garbage, although it is still owned by a thread, and conversely, an ownerless cell can be reclaimed by the ownership on demand mechanism: the examples above are not leaking memory: for instance, the continuation

```
...(e,f)=receive(channel,f);close(e,f)
```

would deallocate everything correctly on the first example. However, would we not allow ownership on demand, ownerless cell would always remain ownerless, and would be a particular case of garbage cells.

Interestingly, ownerless cells are undesirable for Sing#, although it is garbage collected. Execution units in Sing# indeed are processes, and not threads, and can be killed abruptly, so the garbage collector of a process only reclaims the cells that are known to be owned by the process.

There are actually three reasons for creating or supressing ownerless cells:

- at channel's closure, if some messages with non-empty footprints are still in the queue.
- when a message is sent, if the indirect ownership on the reception endpoint is granted by the footprint of the message.
- when a message is received, if the message was not indirectly owned and provides ownership for the reception endpoint a posteriori.

The first case is already prevented by contracts, but the two other cases should be prevented by some condition on the environment proof Γ. Intuitively, this condition should say that no state reachable by a program proved with Γ contain ownerless cells. We however want to have a condition as general as possible, and thus need to define as precisely as possible what reachable states are. Thanks to Lemmas 1, 2, 3, and 5, we already collected quite a lot of restrictions on these reachable states. Let us now formalize a last one, the notion of self-contained state, which will later allow us to give a more general notion of valid environment.

Definition 20 (Self-contained state). *A well-formed open state $\sigma = (\dot{\sigma}, k)$ such that flat$(\sigma) = ((s, h, \dot{k}), \mathsf{emp}(k))$ is said* self-contained *if*

1. *for all ε such that $k(\varepsilon)$ is not the empty queue, $\varepsilon \in dom(\dot{k})$*
2. *for all ε such that $\varepsilon \in dom(\dot{k})$, $\mathsf{perm}(\varepsilon, \dot{k}) = 1$ and $\mathsf{peer}(\varepsilon, \dot{k}) \in dom(\dot{k})$.*

Lemma 11 (Self-containment preservation). *Let p, p', σ, σ' be such that*

- $p, \sigma \xrightarrow[\mathsf{p}]{\Gamma} p', \sigma'$
- $p, \sigma \not\xrightarrow[\mathsf{p}]{\Gamma}$ **error**
- *$\sigma = (\dot{\sigma}, k)$ is well-formed and self-contained.*

Then σ' is self-contained.

Proof. Let EP_0, EP'_0 be the sets of endpoints with permission 1 in σ and σ', EP_1 and EP'_1 the endpoints that are defined in the local state of the flattening of σ and σ', and EP_2, EP'_2 the set of endpoints whose incoming queue is not empty in σ and σ'. We have $EP_2 \subseteq EP_1 \subseteq EP_0$, with EP_1 closed by peer, and we want to show that $EP'_2 \subseteq EP'_1 \subseteq EP'_0$, with EP'_1 closed by peer. We reason by case analysis on the reduction rule

- for non-channel instructions, these sets are unchanged: $EP_0 = EP'_0$, $EP_1 = EP'_1$, $EP_2 = EP'_2$
- for open, $EP'_o = EP_0 \uplus \{\varepsilon, \varepsilon'\}$, $EP'_1 = EP_1 \uplus \{\varepsilon, \varepsilon'\}$, and $EP'_2 = EP_2$.

- for close, $EP_o = EP_0' \uplus \{\varepsilon, \varepsilon'\}$, $EP_1 = EP_1' \uplus \{\varepsilon, \varepsilon'\}$, and $EP_2 = EP_2'$. Moreover, by $p, \sigma, \overset{\Gamma}{\underset{\mathsf{p}}{\not\rightarrow}}$ **MsgError**, $\{\varepsilon, \varepsilon'\} \cap EP_2 = \emptyset$.
- for a send over ε towards ε', $EP_o' = EP_0$, $EP_1' = EP_1$, and $EP_2' = EP_2 \cup \{\varepsilon'\}$. By $p, \sigma, \overset{\Gamma}{\underset{\mathsf{p}}{\not\rightarrow}}$ **OwnError**, we have $\varepsilon \in EP_1$, so $\varepsilon' \in EP_1$.
- for a receive over ε, $EP_o' = EP_0$, $EP_1' = EP_1$, and $EP_2 = EP_2' \cup \{\varepsilon\}$.

□

Let us now define the notion of valid environment. An environment proof is valid if it does not allow to have a subset of the queues whose flattening gives the full permission on the endpoints that are needed to access these queues:

Definition 21 (Valid Environment). *An environment Γ is valid if all contracts in Γ are valid, and for all global state k, if*

- *$(\dot{u}, k)$ is well-formed and self-contained*
- *$\Gamma \vdash k$*
- *$\Gamma \vdash k \rhd \dot{u}$*

then $(\dot{u}, k) = \mathsf{emp}_\emptyset$, e.g. all queues are empty.

Here are several sufficient (but not necessary) conditions that ensure that an environment is valid:

- *à la* Sing#, forbidding the message footprints to contain endpoints that are in a receive state;
- a variant of Sing#: allowing to send server's endpoints only, and allowing only to send them from server's endpoints;
- *á la* Bono et al. [2], imposing a well-foundedness condition.

At this point of the presentation, one might wonder wether the proof environments that were used in the examples are valid. Indeed, some of them are *not* valid. However, using permissions, we will see at the end of the section how to accomodate the proofs so as to use valid environments.

Theorem 2 (Safety). *For all program p, if p is provable safe, then p is safe.*

Proof. Let Γ be a valid environment and p' be a terminating program such that $\vdash_\Gamma \{\mathsf{emp}\}\ p; p'\ \{\mathsf{emp}\}$. We want to show that

1. $p, \mathsf{emp}_G \not\Rightarrow^*$ **error**
2. if $p, \mathsf{emp}_G \Rightarrow^* \sigma$, then σ is leak-free.

Let us start with proving 1. By Theorem 1, $p; p', \mathsf{emp}_G \overset{\Gamma}{\underset{\mathsf{p}}{\not\rightarrow}}^*$ **OwnError**, and $p; p', \mathsf{emp}_G \overset{\Gamma}{\underset{\mathsf{p}}{\not\rightarrow}}^*$ **ProtoError**. By Lemma 9, $p; p', \mathsf{emp}_G \overset{\Gamma}{\underset{\mathsf{p}}{\not\rightarrow}}^*$ **MsgError**, and by definition of $\overset{\Gamma}{\underset{\mathsf{p}}{\rightarrow}}$, $p, \mathsf{emp}_G \overset{\Gamma}{\underset{\mathsf{p}}{\not\rightarrow}}^*$ **error**. Finally, by straightforward induction using Lemma 10, $p, \mathsf{emp}_G \not\Rightarrow^*$ **error**.

Let us now prove 2. Let σ be such that $p, \mathsf{emp}_G \Rightarrow^* \sigma$. Since p' terminates, there is σ' such that $p', \sigma \Rightarrow^*$ skip$, \sigma'$. By definition of $\xrightarrow[\mathsf{p}]{\Gamma}$, $p; p', \mathsf{emp}_G \xrightarrow[\mathsf{p}]{\Gamma}{}^*$ skip$, \sigma'$. By Theorem 1, $\sigma' = (\dot{u}, k')$ for some k'. By Lemmas 2, 3, 5, and 11 (inductively applied), k' satisfies all the conditions of Def 21, thus $\sigma' = \mathsf{emp}_G$. By Lemma 10, $p', \sigma \Rightarrow^* \mathsf{emp}_G$, hence σ is leak-free, which ends the proof. □

Let us now give a new proof of the bounded server that uses a valid environment. Instead of transferring the full ownership on the public endpoint `e_pub`, we assume that the server always keeps half of its ownership, but looses the ownership on the contract's state. It is thus not always allowed to read or update the contract's state of `e_pub`. However, our rules for reflexive ownership transfer allows the client to update the contract's state of `e_pub`, first from 1 to 2 right after it has received `e_pub`, second from 2 to 1 just before sending it back.

```
message con(e) $[e \overset{ep}{\mapsto} C\langle i \rangle * X \overset{ep}{\mapsto} (\overline{S_{1/2}}\langle 1 \rangle, \mathsf{src})]$
message ack $[\mathsf{src} \overset{ep}{\mapsto} \overline{S_{1/2}}\langle 1 \rangle]$
contract S {state 1 {!con->2;} state 2 {?ack->1;}}

server(e_priv) $[e\_priv \overset{ep}{\mapsto} (S\langle 1 \rangle, X) * X \overset{ep}{\mapsto} \overline{S}\langle 1 \rangle]$ {
  local e,f;
  (e,f) = open(C);
  send(con,e_priv,e);
$[e\_priv \overset{ep}{\mapsto} (S\langle 2 \rangle, X) * X \overset{ep}{\mapsto} \overline{S_{1/2}}\langle . \rangle * f \overset{ep}{\mapsto} \overline{C}\langle i \rangle]$
  receive(ack,e_priv);
$[e\_priv \overset{ep}{\mapsto} (S\langle 1 \rangle, X) * X \overset{ep}{\mapsto} \overline{S}\langle 1 \rangle * f \overset{ep}{\mapsto} \overline{C}\langle i \rangle]$
  server(e_priv)||service(f)
} $[\bot]$

client(e_pub) $[\mathsf{emp}]$ {
  local e;
  e=receive(con,e_pub);
$[e\_pub \overset{ep}{\mapsto} S_{1/2}\langle 2 \rangle * e \overset{ep}{\mapsto} C\langle i \rangle]$
  send(ack,e_pub);
$[e \overset{ep}{\mapsto} C\langle i \rangle]$
  ...
}
```

8 Related Works

Turon and Wand [18] were the first to propose a proof system for message-passing concurrency that exploited permissions; their work is based on the π-calculus, focuses on temporal properties and refinement, and is noticeably different from ours. In particular, they only deal with synchronous communications, and do not feature a form of communication contracts.

Leino, Mueller and Smans [12] proposed a proof system for reasoning about asynchronous, unidirectional communication channels and locks. They have no special mechanism for specifying buffer contents, but they can prevent all deadlocks using a mechanism of “levels” of locks and channels, and counting the “debts” and “credits” of channels. These ideas are implemented in the Chalice tool.

Bell, Appel and Walker [1] proposed a proof system for asynchronous unidirectional channels shared by means of permissions, and applied it on one example, the parallelization of a while loop. They do not consider message identifiers, endpoints in the heap, or message safety. They abstract the contents of the buffer by considering the local history, *i.e.* the sequences of messages that were locally observed to transit through the endpoint. In the event of sharing, histories need to be arbitrarily interleaved when two copies of a same shared endpoint are glued together, at the cost of some imprecision and a lack of modularity. Conversely, contracts, being in essence a specialized form of rely/guarantee reasoning, can be considered as more modular. They may moreover be considered as more precise, since the sequences of messages they describe are global, which allows a finer-grained control on how messages emitted from concurrent sources are interleaved.

Bono, Messa, and Padovani [2] accurately pointed out that our previous work made a confusion between orphan messages and memory leaks. Their work, based on the ideas of session types, prompted us to clarify this point in our proof system as well, which we hope we achieved convincingly. Although their work is based on types, there are remarkable similarities with our work. As we already mentioned, they propose a condition on proof environments that is less general than ours. This condition seems influenced by a previous work of Padovani that proposes to view a session type as the projection of the behavior of a program over a given channel [14]. The notion of projection defined by Padovani is however syntax-driven, and due to bound variables, any received channel should not be already owned. Conversely, we adopted a notion of contract obedient programs that is semantics-driven (see Def 13), and do not have to face the problem of bound variables.

Hobor and Gherghina [10] proposed a proof system for synchronization barriers based on separation logic that was helped by “barrier diagrams,” a contract-like description of the sequences of synchronizations on the barrier. The authors conjectured that our previous proof system was not expressive enough to give a proved encoding of barriers. On the contrary, we conjecture that a proved encoding exists if and only if at most N threads are considered and, as we have shown, the encoding can be done using only one channel in our new proof system.

It may look appealing to try to embed some of our proof system in Rely/Guarantee Separation Logic, using concurrent abstract predicates, as recently suggested by Matthew Parkinson [15]. This would replace the definition of an operational semantics by the choice of a particular implementation of queues in the heap, and enable one to reuse the soundness result of concurrent abstract predicates [6]. However, we believe that keeping the actual implementation of queues at an abstract level is more beneficial. For instance, adapting our proof to non-FIFO communications would only amount to a change to the semantics of contracts (see our work on such a theory of contracts [13]). We also believe that, apart from soundness, other results regarding the correctness of communications would not follow as consequences of the soundess of RGSep, and thus a large part of our proofs should be kept.

Sassone, Rathke and Francalanza [8] stated a confluence property for the π-calculus ruled by a strict ownership discipline. Our proof system shows that, with little effort, it is possible to go past confluent programs while still obeying this discipline, but we could not provide a characterization for this new class of concurrent programs.

References

1. C. J. Bell, A. W. Appel, and D. Walker. Concurrent separation logic for pipelined parallelization. In *SAS*, volume 6337 of *LNCS*, pages 151–166, 2010.
2. V. Bono, C. Messa, and L. Padovani. Typing copyless message passing. In G. Barthe, editor, *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2011.
3. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
4. J. Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72, 2003.
5. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.
6. T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. *ECOOP 2010–Object-Oriented Programming*, pages 504–528, 2010.
7. R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In Z. Hu, editor, *APLAS*, volume 5904 of *lncs*, pages 161–177. Springer, 2009.
8. A. Francalanza, J. Rathke, and V. Sassone. Permission-based separation logic for message-passing concurrency. *CoRR*, abs/1106.5128, 2011.
9. M. Giunti and V. T. Vasconcelos. A linear account of session types in the pi calculus. In P. Gastin and F. Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 432–446, 2010.
10. A. Hobor and C. Gherghina. Barriers in concurrent separation logic. In G. Barthe, editor, *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 276–296. Springer, 2011.
11. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138, 1998.
12. K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *ESOP*, volume 6012 of *LNCS*, pages 407–426, 2010.
13. E. Lozes and J. Villard. Reliable contracts for unreliable half-duplex communications. Proceedings of WS-FM'11. To Appear., 2011.
14. L. Padovani. On projecting processes into session types. *Mathematical Structures in Computer Science.*, 2011. To appear.

15. M. J. Parkinson. The next 700 separation logics - (invited paper). In *VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 169–182, 2010.
16. M. Raza and P. Gardner. Footprints in local reasoning. *Logical Methods in Computer Science*, 5(2), 2009.
17. Z. Stengel and T. Bultan. Analyzing singularity channel contracts. In *ISSTA*, pages 13–24, 2009.
18. A. J. Turon and M. Wand. A resource analysis of the pi-calculus. *CoRR*, abs/1105.0966, 2011.
19. J. Villard. *Heaps and Hops*. PhD thesis, École Normale Supérieure de Cachan, february 2011.
20. J. Villard, É. Lozes, and C. Calcagno. Proving copyless message passing. In *APLAS*, volume 5904 of *LNCS*, pages 194–209, Seoul, Korea, Dec. 2009.