Shared contract-obedient channels

Etienne Lozes, Jules Villard

Science of Computer Programming 100 (2015) 28–60

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Shared contract-obedient channels

Étienne Lozes [a,*], Jules Villard [b]

[a] Universität Kassel, Germany
[b] University College London, UK

ARTICLE INFO

Article history:
Received 15 January 2013
Received in revised form 24 May 2014
Accepted 22 September 2014
Available online 2 October 2014

Keywords:
Program verification
Message passing
Linear channels
Separation logic

ABSTRACT

Recent advances in the formal verification of message-passing programs are based on proving that programs correctly implement a given protocol. Many existing verification techniques for message-passing programs assume that at most one thread may attempt to send or receive on a channel endpoint at any given point in time, and expressly forbid endpoint sharing. Approaches that do allow such sharing often do not prove that channels obey their protocols. In this paper, we identify two principles that can guarantee obedience to a communication protocol even in the presence of endpoint sharing. Firstly, threads may concurrently use an endpoint in any way that does not advance the state of the protocol. Secondly, threads may compete for receiving on an endpoint provided that the successful reception of the message grants them ownership of that endpoint retrospectively. We develop a program logic based on separation logic that unifies these principles and allows fine-grained reasoning about endpoint-sharing programs. We demonstrate its applicability on a number of examples. The program logic is shown sound against an operational semantics of programs, and proved programs are guaranteed to follow the given protocols and to be free of data races, memory leaks, and communication errors.

0. Introduction

Message-passing idioms appear everywhere in today's software: from the Message Passing Interface (MPI) used in high-performance computing, to the inter-process communication layer in Android apps, and to Web services. As for other forms of concurrency, naively checking the correctness of a message-passing system is severely impaired by the combinatorial explosion of the number of possible interactions between the components of the system. One way to tackle this issue is to develop formal verification techniques for message-passing programs, such that reasoning about a system is tantamount to reasoning about each component in isolation. A promising avenue in this respect is to separate the study of programs from the study of the protocols they are meant to implement, i.e. prove that a program correctly implements a protocol on the one hand, and reason about that protocol independently of its implementation on the other hand. In this context, protocols act both as specifications of what a program is allowed to do and as descriptions of the actions that programs must expect from the environment. Two main approaches coexist for describing such protocols: session types on the one hand [30], used to police interactions in programs expressed either in the π-calculus [20] or in a message-passing variant of Java [21], and *channel contracts* on the other hand [7], used for instance to describe the protocols in the Sing♯ programming language [13] developed for the Singularity operating system [22]. High-level protocol descriptions such as these allow the program verification effort to be split between checking properties at the level of the protocol itself on the one hand, and checking obedience of each thread of the program to its part in the protocol on the other hand. If all threads play their parts

* Corresponding author.

http://dx.doi.org/10.1016/j.scico.2014.09.008
0167-6423/

according to the protocol, then the program as a whole inherits the good properties of that protocol. This idea underpins many analyses of message-passing programs; it has been made more explicit in recent works both on channel contracts [32] and on session types [11].

Most of the existing verification techniques for message-passing programs assume that channel endpoints are used in a linear fashion: no two threads may ever try to send or receive simultaneously on the same channel endpoint. This allows the checking of the conformance to a given protocol to be local to each thread or process. Without this restriction, different threads sharing the same endpoint might have discordant views about at which point in the protocol that endpoint is, which greatly hinders the proof that threads indeed abide by that protocol. However, imposing that endpoints are used linearly reduces the scope of these techniques, as many useful paradigms require some form of endpoint sharing between processes. Moreover, this restriction enforces some form of determinism on programs, which excludes the encoding of standard synchronisation primitives such as locks and semaphores [14].

This work presents a program logic for message-passing programs that may share channel endpoints, while giving meaningful protocols to their interactions. The program logic achieves two goals: on the one hand, checking that the exchange of messages on the channels used by programs obeys protocols defined by *channel contracts*, a particular representation of protocols as communicating finite state machines; on the other hand, ensuring the absence of data races and resource leakage. The program logic allows two forms of sharing on channel endpoints.

The first form of sharing allows threads to use endpoints concurrently as long as they do not advance the protocol state. This ensures that a consistent view of the contract state is maintained amongst all sharers of a given endpoint. This form of sharing is useful when shared endpoints exchange only one kind of messages, and unidirectionally, as long as the sharing subsides. This is the case for instance when one or several producers send the same kind of message repeatedly over a channel, to be received by one or several consumers.

The second form of sharing allows several threads to compete for reception of a message on a shared endpoint. Exclusive access to this very endpoint is granted to the winner, who can then use it to realise the rest of the communication. Meanwhile, the other threads are kept waiting for the initial message until the protocol comes back to the initial state and the ownership of the endpoint is released by the winning thread. This form of sharing is typical of two situations occurring in existing message-passing applications. First, it can occur when several worker threads or processes listen for the same kind of event, each individual event being picked up by one worker only. This is the case for instance for *implicit intents* in the Android framework [1], where components can register as able to provide certain services, later to be called by applications in need of these services. For instance, web browsers register as being able to process intents of the “browsable” category. Clicking on a web link in an email emits an implicit intent of that category, which can be picked up by the web browser. In the case of Android, such sharing results in simple protocols: either a single intent is sent as in the example above, or the intent awaits a response. We show that, in fact, any protocol can be realised on the shared endpoint once it has been acquired in such a way. Surprisingly, sharing in this way does not contradict linear channel usage: each endpoint is effectively used by at most one thread at a time even though several threads compete for the initial message. More generally, this work shows how linearity can be relaxed to get both expressive protocols and sharing. In doing so, we open the way for bringing more forms of sharing to new message-passing run-time libraries based on formal methods in general, such as session types or Sing♯, which are currently more strict in their enforcing of linearity.

This work builds on a previous approach based on a marriage of separation logic and channel contracts [33], which forbade any active sharing of endpoints. As in previous work [26], we consider a simple imperative language that features primitives for dynamically creating and destroying bi-directional, asynchronous channels, each made of two endpoints, and for sending and receiving messages on individual endpoints. Each message is composed of a user-defined tag (which can be used to describe the kind of payload supposed to be transferred, as in Sing♯ or MPI) and zero or more values. Crucially, values may be references to other endpoints, and thus sending a message may create sharing of resources and foster concurrency errors. In the absence of endpoint sharing, previous work was able to prove obedience to channel contracts and absence of data races, and from that to deduce the absence of message reception errors and of orphan messages. A key ingredient of the program logic, which we have retained, is to logically reflect the transfer of a message that is attached to a resource by the transfer of the ownership of that resource (the message’s *footprint* in memory) to the recipient of the message. Contrarily to similar transfer disciplines such as session types, footprints need not be syntactically determined by the value sent: sending the address of an endpoint over a channel does not equate sending the ownership of that endpoint. Rather, any footprint may be attached to the message. This makes our approach more powerful in terms of which programs can be proved correct, as it allows more complex ownership transfer disciplines.

The contributions of the present paper are as follows:

- We identify two patterns for sharing endpoints while retaining the ability to describe their interactions via meaningful protocols.
- We introduce a new program logic able to prove programs that abide by these patterns. The presentation of the program logic unifies both sharing paradigms.
- We justify the soundness of our program logic by proving it sound with respect to an operational semantics. Our soundness proof is able to express the fact that proved programs obey their contracts and are free of communication errors, by linking the semantics of programs to that of the channel contracts they implement.

Outline In the first section we define our programming language featuring bi-directional, dynamically allocated, asynchronous channels, and introduce the syntax of contracts. We illustrate contract obedience in the presence of sharing on examples. In the second section, we give an overview of separation logic and contracts as used in our own program logic, introduced in the third section. We sketch the verification of a few telling examples in a fourth section. In the fifth section, we provide an operational semantics for channel contracts and for our programming language. In the sixth section, we establish the soundness of our program logic with respect to this semantics. We conclude with related works.

1. Programming language

1.1. Syntax

We consider an idealised imperative programming language with support for message-passing concurrency. Threads share a global memory and may allocate, deallocate, and manipulate heap objects in this shared memory. For the purpose of this work, it is enough to limit heap objects to *channels*. Communication channels are asynchronous, bi-directional, and each made of two *channel endpoints* (or simply endpoints): each endpoint can be used for sending to and receiving from the other endpoint. A channel can hence be implemented as a pair of FIFO buffers allocated on the heap: each endpoint receives messages by dequeuing them from one of the buffers, and sends messages by enqueuing them into the other. This communication model is close to that of Sing♯ [13]. Two endpoints of the same channel are called *peers*.

We assume infinite sets $\mathtt{Var} = \{\mathtt{e}, \mathtt{f}, \mathtt{x}, \mathtt{y}, \ldots\}$, $\Sigma = \{m, \ldots\}$, and $\mathtt{Val} = \{v, \ldots\}$ of respectively variables, message identifiers (or *tags*), and values. The grammar of expressions, boolean expressions, atomic commands and programs is as follows:

$$E ::= \mathtt{x} \mid v \mid E_1 + E_2 \mid \cdots \qquad \text{expressions}$$

$$B ::= E = E \mid B \texttt{ and } B \mid \texttt{not } B \qquad \text{boolean expressions}$$

$$c ::= \texttt{skip} \mid \texttt{assume}(B) \mid \mathtt{x} = \mathtt{E} \qquad \text{atomic commands}$$

$$\mid (\mathtt{e}, \mathtt{f}) = \texttt{open}() \mid \texttt{close}(\mathtt{e}, \mathtt{f}) \mid \texttt{send}(\mathtt{m}, \mathtt{e}, E_1, \ldots, E_n) \mid (\mathtt{x}_1, \ldots, \mathtt{x}_n) = \texttt{receive}(\mathtt{m}, \mathtt{e})$$

$$\mathit{cases} ::= \_ \mid (\mathtt{x}_1, \ldots, \mathtt{x}_n) = \texttt{receive}(\mathtt{m}, \mathtt{e}) : p \ \mathit{cases}$$

$$p ::= c \mid p;\ p \mid p \parallel p \mid p + p \mid p^* \mid \texttt{local x in } p \mid \texttt{switch}\ \{\ \mathit{cases}\ \} \qquad \text{programs}$$

The `skip` command does nothing; `assume(`B`)` does nothing if B holds and blocks otherwise (thus `skip` is equivalent to `assume(`*true*`)`); `x = `E evaluates the expression E and stores the result in the variable `x`; `(e,f) = open()` creates a new channel and allocates both of its associated endpoints `e` and `f` in the heap with empty buffers; `close(e,f)` disposes the endpoints at addresses `e` and `f`; `send(m,e,`$E_1, \ldots, E_n$`)` sends a message over the endpoint `e` to its peer endpoint; the message has a tag `m`, and a tuple $E_1, \ldots, E_n$ of n message values ($n \geq 0$); `(x`$_1, \ldots,$`x`$_n$`) = receive(m,e)` receives the first available message on endpoint `e`, which must have tag `m` and arity n, and sets the variables `x`$_1$, ..., `x`$_n$ according to the payload of the message. If no message is available, or if the first available message has a tag different than `m`, the command blocks; the sequential composition of commands is written $p_1; p_2$; $p_1 \parallel p_2$ is the parallel composition; $p_1 + p_2$ is the internal choice; p^* is the Kleene iteration; `local x in` p is the creation of a local variable. The `switch` construct is used to wait for several messages on several endpoints at once. The construct selects a branch corresponding to an available message if possible. If the message buffers of all the endpoints in the `switch` construct are empty, then it blocks. If a message is available on one of these endpoints, but with a tag that has no corresponding `receive` branch, then it triggers an error. Let us illustrate the behaviour of `switch` with an example.

Example 1. Consider the following code snippet:

```
switch {
  x = receive(bool, e): {p1}
  x = receive(int, e): {p2}
  (y,z) = receive(pair, d): {p3} }
```

The following scenarios are possible results of executing the code above:

- No message ever arrives on `e`, and this thread is stuck.
- The message $(\mathtt{int}, n)$ is next in line at endpoint `e`: the message is dequeued, `x` is set to n, and p_2 is executed.
- The messages $(\mathtt{int}, n)$ and $(\mathtt{pair}, v_1, v_2)$ are available on respectively `e` and `d`: either of them can be picked up and, accordingly, either p_2 or p_3 is run next (with respectively $\mathtt{x} = n$ or $\mathtt{y} = v_1$ and $\mathtt{z} = v_2$).
- A message that is tagged with neither `bool` nor `int` is available on `e`, for instance a message $(\mathtt{float}, 4.52)$: an error is raised.

One can define the usual `if` and `while` constructs of programming languages using `assume` and non-deterministic choice in a standard way. Thus, we write `if (`B`)` p_1 `else` p_2 for

`(assume(`B`);` p_1`) + (assume(not` B`);` p_2`).`

We sometimes write `if (*) then` p_1 `else` p_2 instead of $p_1 + p_2$ to represent internal choice.

Likewise, `while (`B`)` p is syntactic sugar for

`(assume(`B`);` p`)*; assume(not` B`).`

Example 2. The following program allocates a channel, stores the two endpoints in global variables `e` and `f` and spawns two threads, `put` and `get`, which exchange either an integer (with tag `int`) or a boolean (with tag `bool`). The tag of the message is used to inform the `get` process of the choice that has been made.

```
main() {
  (e,f) = open();
  put() || get();
  close(e,f);
}
```

```
put() {
  if (*) {
    send(int,e,32);
  } else {
    send(bool,e,1);
  }
}
```

```
get() {
  switch {
    n = receive(int, f): { ... }
    b = receive(bool, f): { ... }
  }
}
```

In most of our examples including this one, we use procedures for better readability of the code. Procedures could be added to the language and the program logic using standard techniques. Moreover, our examples will not use recursive procedures, so procedures can always be inlined.

1.2. Contracts

In this section, we introduce *contracts*. Contracts are used to describe the protocol followed by each communication channel in programs. A contract is a finite automaton whose transitions are labelled either with send $!m$ or with receive $?m$ actions. The paths in the automaton describe the admissible sequences of emissions and receptions (looking only at the tag m of each message) on a channel. The final states represent points in the protocol when the channel is allowed to be closed.

We assume an infinite set `Control` of control states. Recall that Σ is the set of message identifiers.

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

A label $\lambda \in \{!, ?\} \times \Sigma$ in the contract automaton is written either as a send label $!m$ or as a receive label $?m$. Given a contract $C = (Q, \Sigma, \delta, q_0, F)$, we write $\mathit{init}(C)$ for q_0, $\mathit{finals}(C)$ for F, and $q \xrightarrow{\lambda} q' \in C$ if $\delta(q, \lambda)$ is defined and equal to q'.

In this work, we consider bi-partite channels made of two endpoints that follow *dual* contracts. We describe their interaction as a single contract C written from the point of view of the first endpoint; the other endpoint implicitly follows the dual contract $\bar{C}$, where sends ! and receives ? have been swapped.

Definition 4 *(Dual of a contract).* The dual $\bar{\lambda}$ of a label λ is defined as the involution $\overline{!m} = ?m$ and $\overline{?m} = !m$. The dual of a contract $C = (Q, \Sigma, \delta, q_0, F)$ is the contract $\bar{C} = (Q, \Sigma, \bar{\delta}, q_0, F)$ such that $\bar{\delta}(q, \lambda)$ is defined when $\delta(q, \bar{\lambda})$ is, and $\bar{\delta}(q, \lambda) = \delta(q, \bar{\lambda})$.

A contract and its dual describe the behaviour of a channel in a program, abstracting away from the exact values exchanged to focus solely on the tags of messages. We are interested in establishing the absence of two kinds of error in the interaction between a contract and its dual: (1) *unspecified receptions*, where a message carrying an unexpected tag is received, and (2) *orphan messages*, i.e. messages that have been sent but not received at the time a channel is closed. Contracts C that exhibit neither of these errors in all the possible executions of C together with its dual $\bar{C}$ are called *valid*.

We delay the formal definition of the semantics of contracts and thus the definition of validity until Section 5.1. For now, it suffices to know that sufficient, syntactic conditions exist that ensure that a contract is valid [18,25]. Let us recall them here.

Definition 5 *(Well-formed contracts).* A contract is *polarised* if there are no q, q_1, q_2, m_1, m_2 such that $q \xrightarrow{!m_1} q_1$ and $q \xrightarrow{?m_2} q_2$. A cycle is a sequence $q_0 \xrightarrow{\lambda_1} q_1 \xrightarrow{\lambda_2} \dots \xrightarrow{\lambda_k} q_k$ such that $k \geq 1$ and $q_k = q_0$. A state is *synchronising* if all cycles going through this

[1] We could allow non-deterministic automata as well, i.e. allow δ to be of the type $Q \times \{!, ?\} \times \Sigma \rightharpoonup \mathcal{P}(Q)$, at the cost of complicating the formalism. Moreover, determinism can always be assumed by standard automaton determinisation.

state contain at least a send and a receive transition. A contract that is polarised and whose final states are synchronising is *well-formed*.

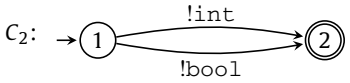
All the contracts we will use as program specification in our examples are well-formed.

1.3. Contract-obedient communications and sharing

In this section, we informally introduce the notion of contract obedience via several examples (with and without endpoint sharing) and their associated contracts.

Contracts naturally describe the allowed sequences of send and receive actions on a given channel, abstracting away from the actual values carried by messages to retain only tags. Upon creation, endpoints are assigned a contract in the form of an annotation to the `open()` command. In our program logic, introduced formally in Section 3, the `(e,f) = open(`C`)` instruction creates endpoints pointed to by `e` and `f`, which follow contracts C and $\bar{C}$ respectively, and start in the initial state q_0 of C. To a `send(m,e,v)` instruction must correspond a $!m$ transition in the contract, and similarly for `receive` instructions. A `switch`/`receive` block must account for all possible tags that may be received in the current state of the contract (for each receiving endpoint). The `close(e,f)` instruction must only be called when both endpoints are in the same final state of the contract. Following such a discipline ensures the absence of communications errors in programs, provided that contracts are well-formed.

For instance, one can easily check that the program in Example 2 obeys the following two-state contract:



Now, consider

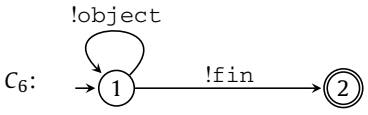
The `put` thread (and thus the whole program) violates contract C'_2 since `put` may also send a `bool` message which is not a valid transition in C'_2. Conversely, the `get` thread violates contract C''_2 since its `switch`/`receive` block is not ready for `float` messages that C''_2 prescribes. On the other hand, `put` obeys C''_2, since not all sending transitions of the contract have to be realised by the program, and `get` obeys C'_2, since being ready for more tags than necessary does not impact safety. There is thus an asymmetric treatment of what it means to obey the contract when sending and when receiving.

Example 6. The following program implements a classic producer/consumer scenario: the `main()` function allocates a channel and spawns two threads, `producer` and `consumer`. The former sends objects to the latter for some time via `object`-tagged messages. To signify the end of the communication, a `fin`-tagged message is sent. Upon receiving `fin`, `consumer` closes the channel.

```
main() {
  (e,f) = open();
  producer() || consumer();
}

producer() {
  while (*) {
    ... /* produce an object x */
    send(object,e,x);
  }
  send(fin,e);
}
```

```
consumer() {
  done = 0;
  while(not done) {
    switch {
      y = receive(object, f):
        { ... /* do something with y */ }
      receive(fin, f): { done = 1; }
    }
  }
  close(e,f);
}
```

This program implements the following contract:



Note that once the `fin` message has been exchanged, both endpoints are in the same final state 2 of C_6, thus it is valid (from the contract's point of view) to close the channel at this point.

Example 7. We can extend the previous example to the case of several producers, which all share one end of the channel. The `fin` message is sent to the consumer once all producers have finished sending objects. Note that, without more information about the number of objects that will be sent over the channel, another version of this program with several consumers running in parallel would require additional message exchanges to inform all the consumers of the end of the communication.

```
main() {
  (e,f) = open();
  producers() || consumer();
}

producers() {
     producer()
  || ...
  || producer();
  send(fin,e);
}
```

```
producer() {
  while (*) {
    ...
    send(object,e,x);
  }
}
```

```
consumer() {
  done = 0;
  while(not done) {
    switch {
      y = receive(object, f): { ... }
      receive(fin, f): { done = 1; }
    }
  }
  close(e,f);
}
```

This program also obeys the contract C_6, essentially thanks to the self-loop at state 1, which allows the multiple senders to use the endpoint without invalidating each other's view of the endpoint's contract state.

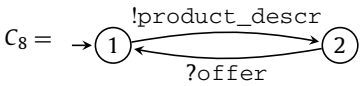
Example 8. The following example showcases another scenario where endpoint sharing arises naturally. Consider a seller and several buyers running in parallel. The seller advertises its product using endpoint `e` via a `product_descr` message to several buyers. The buyers concurrently access the peer `f` of `e` and make offers. The first buyer to make a suitable offer wins the auction. To keep things simple, we do not include a mechanism for closing the channel at the end of the interaction between the seller and the buyers. Thus, the channel is never closed.

```
seller(e) {
 local price = 0;
 while (!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);
}
```

```
main {
 (e,f) = open();
 seller(e) ||
  buyer(f) ||
  ... ||
  buyer(f);
}
```

A natural contract for the communication channel `(e, f)` is



Remarkably, none of the buyers makes any assumption on the contract state of `f` when they simultaneously try to receive `product_desc`. In particular, they attempt to receive a product description on `f` without the knowledge that `f` is in a contract state that ensures that such a message will be available next. The crucial argument that validates this program is that the received message justifies *in hindsight* the buyer waiting for a `product_descr` message. In other words, the knowledge of the contract state of `f` is established after receiving on `f`. The thread that successfully receives the endpoint has to follow through with the rest of that endpoint's protocol. In this example, it will send back an offer, thereby completing the loop in the protocol. This form of sharing differs significantly from that of the previous example in the arguments that justify contract obedience.

1.4. Program safety

Our program logic will establish the absence of the following kinds of run-time errors in programs, which we will formalise in Section 6.2:

Ownership errors They occur when a thread tries to deallocate an endpoint at the same time another thread does any other operation on this same endpoint (a send, a receive, or a deallocation).

Memory leaks They occur when the memory state becomes such that no continuation of the program can deallocate all resources, in our case all channel endpoints.

Orphan messages They occur when a channel with pending messages is closed.

Unspecified receptions They occur when a thread enters a `switch/receive` 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/receive`.

Contract violation They occur when a program doesn't abide by one of its channel contracts: a message with an unexpected tag is sent, or not enough tags are accounted for in a `switch/receive` according to the contract, or the channel is closed when endpoints are not in the same final state.

Example 9. The program

$$P_0 \triangleq \texttt{send}(\texttt{m}_1, \texttt{e}) \;||\; \texttt{switch}\ \{\ \texttt{receive}(\texttt{m}_2, \texttt{f})\texttt{: skip;}\ \}$$

triggers an unspecified reception. Replacing `get()` by `skip` in Example 2 would cause a contract violation (the endpoint `f` is still in the initial, non-final state when the channel is closed) as well as an orphan message. Replacing `consumer()` by `skip` in Example 6 would leak both endpoints of the communication channel.

Note that all of the errors we listed are incomparable, although it is possible for single program to exhibit several of them. In particular, orphan messages are not a special case of memory leaks: it may be the case that lost messages carried no heap data (e.g. messages that consist of scalar values such as integers or booleans).

2. Proof principles for contract-obedient sharing of endpoints

In this section, we introduce the main ideas that underpin our new program logic. This logic marries separation logic on the one hand and channel contracts on the other hand, and establishes the absence of the errors listed in Section 1.4 in programs, as well as contract obedience. Program specifications established by our program logic take the form of Hoare triples $\{\varphi\}\ p\ \{\psi\}$. Hoare triples are understood in terms of partial correctness, that is, a proved program is not guaranteed to terminate. Separation logic naturally enforces some properties such as data race freedom and linear usage of endpoints, as we will see in Section 2.1. The challenge is to use separation logic to check contract obedience in the presence of endpoint sharing, and from that to deduce the absence of orphan messages and unspecified receptions in proved programs.

A notable property not guaranteed by our program logic is deadlock-freedom. We believe that deadlocks are a concern orthogonal to the main aim of this paper, which is to verify contract obedience of channel-sharing programs, and that our program logic can be extended to prevent deadlocks by borrowing from existing techniques, for instance session types [20] or Leino et al. [24].

2.1. Ownership and separation

Separation logic enforces and exploits locality principles in programs. As a first approximation (before we introduce permissions), these locality principles can be summarised as follows.

Ownership hypothesis Each thread owns a region of the heap: it can only read and update that part of the heap. The limits of the heap region owned by a thread may however evolve during the execution.

Separation property At any point in time, each heap object is owned by at most one thread. In the context of message passing, each *allocated* endpoint is owned either by exactly one thread or by a message currently stored in another endpoint's queue.

Before we introduce our assertion language and program logic formally, let us illustrate these principles with proof sketches of the programs of Examples 2 and 6. Once our program logic has been introduced, the reader is invited to come back to these proof sketches and check that they are indeed derivable in our framework.

The proof sketches, shown in Fig. 1, consist of annotating programs with ownership information. An annotations is written in brackets `[`φ`]` and denotes the fact that φ holds at that program point (e.g. in the case of a loop, φ is the loop invariant). We give the formal syntax of formulas in Section 3.1. In the meantime, let us briefly and incompletely describe it. A predicate $x \mapsto (C\langle q\rangle, y)$ denotes the ownership of the endpoint at address x, whose peer is y, and which follows contract C and is currently in the state q of that contract. The predicate emp denotes the empty heap. The connectives that can be seen in Fig. 1 are those of classical logic, except for the separating conjunction $*$ of separation logic: $\varphi * \psi$ is true of a state if its resources (the endpoints in our case) can be split into two *disjoint* sets of resources, such that one sub-state satisfies φ and the other satisfies ψ.

In both programs shown in Fig. 1, the precondition of `main`, emp, indicates that nothing is known to be allocated (or, alternatively, that nothing is owned) at the beginning of the execution. The main thread first opens a new channel, after which it owns both of its endpoints, whose addresses have been stored in variables `e` and `f` respectively. Here the separating conjunction $*$ is used to add together the two disjoint pieces of owned heap, each consisting of a single endpoint. Each thread then gets a different endpoint (in accordance with the separation property).

```
main() [emp] {
  (e,f) = open();
  [e ↦ (C2⟨1⟩, f) ∗ f ↦ (C̄2⟨1⟩, e)]
  [e ↦ (C2⟨1⟩, f)] || [f ↦ (C̄2⟨1⟩, e)]
  put() || get();
  [e ↦ (C2⟨2⟩, f)] || [f ↦ (C̄2⟨2⟩, e)]
  [e ↦ (C2⟨2⟩, f) ∗ f ↦ (C̄2⟨2⟩, e)]
  close(e,f);
} [emp]

put() [e ↦ (C2⟨1⟩, f)] {
  if (∗) {
    send(int,e,32);
  } else {
    send(bool,e,1);
  }
} [e ↦ (C2⟨2⟩, f)]

get() [f ↦ (C̄2⟨1⟩, e)] {
  switch {
    n = receive(int, f): { [f ↦ (C̄2⟨2⟩, e)] ... }
    b = receive(bool, f): { [f ↦ (C̄2⟨2⟩, e)] ... }
  }
} [f ↦ (C̄2⟨2⟩, e)]
```

```
main() [emp] {
  (e,f) = open();
  [e ↦ (C6⟨1⟩, f) ∗ f ↦ (C̄6⟨1⟩, e)]
  producer() || consumer();
  [emp ∗ emp]
} [emp]

producer() [e ↦ (C6⟨1⟩, f)] {
  while (∗) [e ↦ (C6⟨1⟩, f)] {
    ...
    send(object,e,x);
  }
  [e ↦ (C6⟨1⟩, f)]
  send(fin,e);
} [emp]

consumer() [f ↦ (C̄6⟨1⟩, e)] {
  done = 0;
  while(not done)
[(done = 0 ∧ f ↦ (C̄6⟨1⟩, e))
 ∨(done = 1 ∧ f ↦ (C̄6⟨2⟩, e) ∗ e ↦ (C6⟨2⟩, f))] {
    switch {
      y = receive(object, f): { ... }
      receive(fin, f): { done = 1; }
    }
  }
  [f ↦ (C̄6⟨2⟩, e) ∗ e ↦ (C6⟨2⟩, f)]
  close(e,f);
} [emp]
```

Fig. 1. Ownership annotations for Examples 2 and 6.

In Example 2, `put` starts with precondition $e \mapsto (C_2\langle 1\rangle, f)$ and `get` starts with $f(\bar{C}_2\langle 1\rangle e)$. Each thread retains ownership of its endpoint for its entire execution. The `put` thread sends either an `int` or a `bool` message; whatever branch ends up being executed, the endpoint will be in state 2 at the end. Conversely, the `get` thread has to be ready for any of the possibilities indicated by $\bar{C}_2$, lest an unexpected reception occurs, and finishes with `f` in state 2 of $\bar{C}_2$. The main thread continues by collecting the postconditions of each thread after they have both finished executing. In this case, this yields the two endpoints in state 2 of C_2 and $\bar{C}_2$, respectively. Since 2 is a final state of C_2, the channel can be safely closed.

Behind the scenes, the proof of the parallel composition of `put` and `get` in the main thread uses the following rule of concurrent separation logic, which justifies distributing disjoint pieces of the heap to each thread and merging the corresponding postconditions at the end:

$$\textsc{Parallel}\quad \frac{\{\varphi\}\ p\ \{\psi\} \qquad \{\varphi'\}\ p'\ \{\psi'\}}{\{\varphi * \varphi'\}\ p \parallel p'\ \{\psi * \psi'\}}$$

At first glance, this rule seems to prevent two threads from using the same resources. However, as we are about to see with the proof sketch of Example 6, sending and receiving messages allow pieces of heap to be logically transferred from one thread to another.

The proof sketch of Example 6 is similar to that of Example 2, except for the treatment of the `fin` message. Indeed, the ownership of endpoint `e` is *transferred* from the producer thread to the consumer when the `fin` message is exchanged. This justifies why `consumer` can safely close the channel, even though `producer` initially owns the endpoint `e`. In the program logic, each message tag is assigned a separation logic formula, called its *footprint*, which is logically lost upon sending messages with that tag, and gained upon receiving. In this case, the footprint of the message is described by the formula $e \mapsto (C_6\langle 2\rangle, f)$: when the `producer` thread gives up ownership of `e`, `fin` has been sent and thus `e` is in state 2 of C_6, which is the state in which `consumer` receives it to close the channel.

2.2. Beyond linearity with permissions

At first glance, the separation property and the Parallel rule seem at odds with endpoint sharing: for two threads to be using the same endpoint concurrently, both have to own it. However, the formula $e \mapsto (C\langle q\rangle, f) * e \mapsto (C\langle q\rangle, f)$ is inconsistent, as it implies e.g. $e \neq e$, thus preventing an obvious application of the Parallel rule. A first way around that apparent limitation is to introduce *fractional permissions* [6,5]. The permission of an owned piece of heap is any quantity

```
producers() [e ↦ (C6⟨1⟩, f)] {
  [e ⇛.5 (C6⟨1⟩, f) ∗ e ⇛.5 (C6⟨1⟩, f)]
  producer() || producer();
  [e ⇛.5 (C6⟨1⟩, f) ∗ e ⇛.5 (C6⟨1⟩, f)]
  [e ↦ (C6⟨1⟩, f)]
  send(fin,e);
} [emp]
```

```
producer() [e ⇛.5 (C6⟨1⟩, f)] {
  while (*) {
    ...
    /* produce an object and stores things in x */
    send(object,e,x);
  }
} [e ⇛.5 (C6⟨1⟩, f)]
```

Fig. 2. Fractional shares for endpoint sharing: proof sketch of Example 7. The proofs of the `main` and `consumer` functions are identical to those shown in Fig. 1 and are not repeated here.

$\pi \in (0, 1]$.[2] The permission 1 is the "full" permission while a permission $\pi < 1$ is a "partial" permission. To denote fractional ownership of an endpoint, we add a subscript to the predicate: $e \Rightarrow_\pi (C\langle q\rangle, f)$ denotes the ownership of a fraction π of the endpoint e. We omit the subscript when it is 1 (thus, $e \Rightarrow_1 (C\langle q\rangle, f) \Leftrightarrow e \mapsto (C\langle q\rangle, f)$). Crucially, an endpoint with permission π such that $0 < \pi = \pi_1 + \pi_2 \leq 1$ can be split into two disjoint pieces of ownership with permissions π_1 and π_2, as expressed by the following logical equivalence (where $\oplus$ is the addition operation on the reals defined only when the sum is less or equal to 1):

$$e \Rightarrow_{\pi_1 \oplus \pi_2} \left(C\langle q\rangle, f\right) \quad \Leftrightarrow \quad e \Rightarrow_{\pi_1} \left(C\langle q\rangle, f\right) * e \Rightarrow_{\pi_2} \left(C\langle q\rangle, f\right)$$

Holding an endpoint with the full permission means that no other thread is currently using that endpoint. Hence, the full permission grants unrestricted access to the endpoint, for sending, receiving, or closing the channel. Holding only a partial permission means that other threads might be accessing the endpoint concurrently. Hence, in order not to invalidate their view of the endpoint's state, a partially owned endpoint can only be used for communications that *do not update* the contract state, and closing a channel with partially owned endpoints is forbidden.

This additional expressive power allows us to prove the multiple producers program from Example 7. A sketch of the proof is shown in Fig. 2. Note that allocating a new channel creates endpoints with the full permission. The permission is split at the level of the parallel composition in `producers` to distribute fractions of the endpoint `e` to each producer thread. Each producer is then allowed to send `object` messages thanks to the self-loop on state 1 of C_7, but not the `fin` message, as this would update the contract state to 2, which is forbidden by the permission discipline. After all the producers have finished running, all partial permissions are collected back into the full permission, allowing `putters` to send the `fin` message with footprint $\mathtt{e} \mapsto (C\langle 2\rangle, \mathtt{f})$ to the consumer thread, which combines it with its own to close the channel.

2.3. Linearity in hindsight

Our second sharing paradigm relies on a specific discipline of ownership transfer, whereby the entire ownership of the receiving endpoint is transferred during the message initiating the communication. In particular, the receiver thread initially holds no ownership on that endpoint. Receiving the initial message grants it both the *a posteriori* knowledge of the state of the endpoint and the right to continue the communication in accordance to the contract.

This principle is illustrated informally in the proof sketch of Example 8 shown in Fig. 3. In this example, the footprint associated to the `product_descr` message is $\mathtt{f} \mapsto (\bar{C}_8\langle 1\rangle, \mathtt{e})$ (note, and the one associated to the `offer` message is $\mathtt{f} \mapsto (\bar{C}_8\langle 1\rangle, \mathtt{e})$). Note that these footprints bear no syntactic relationship with the values of the corresponding messages, illustrating the expressivity of footprint-based ownership transfer.

Although straightforward in its application, the reason why linearity can soundly be established in hindsight is all but immediate. One could imagine a buyer receiving `f` while another buyer is in the middle of realising its interaction on the same `f`, thus violating both contract obedience (from the point of view of the seller, who does not expect two "sessions" to be running concurrently), and linearity. In this example, it is clear why this cannot happen: `product_descr` message is only sent at the beginning of each session. More generally, to guarantee that messages that grant ownership of the recipient endpoint a posteriori can only be sent once per session, we need a carefully crafted semantics of programs that keeps track of what is owned by each thread (see Section 5.3).

3. Program logic

In this section, we formally introduce the syntax and semantics of the assertions of the program logic, followed by its proof rules.

[2] For simplicity, we restrict our attention to fractional permissions, but any other permission model could be used instead, such as tokens [5] or tree shares [12].

```
seller(e) [e ↦ (C8⟨1⟩, f) ∗ f ↦ (C̄8⟨1⟩, e)] {
 local price = 0;
 while (!good(price)) [e ↦ (C8⟨1⟩, f) ∗ f ↦ (C̄8⟨1⟩, e)] {
   send(product_descr,e);
   [e ↦ (C8⟨2⟩, f)]
   price = receive(offer,e);
 }
}

main [emp] {
 (e,f) = open();
 [(e ↦ (C8⟨1⟩, f) ∗ f ↦ (C̄8⟨1⟩, e)) ∗ emp ∗ ··· ∗ emp]
 seller(e) || buyer(f) || ... || buyer(f);
 [(e ↦ (C8⟨1⟩, f) ∗ f ↦ (C̄8⟨1⟩, e)) ∗ emp ∗ ··· ∗ emp]
 [e ↦ (C8⟨1⟩, f) ∗ f ↦ (C̄8⟨1⟩, e)]
 close(e,f);
} [emp]
```

```
buyer(f) [emp] {
 local x;
 receive(product_descr,f);
 [f ↦ (C̄8⟨2⟩, e)]
 x = think_about_it();
 send(offer,f,x);
} [emp]
```

Fig. 3. Linearity in hindsight: proof of Example 8.

3.1. Syntax and semantics of assertions

Syntax Assertions convey information about the values of the program variables and the resources that are owned by a thread at a given point in time, and the values that are transmitted and the resources whose ownership is transferred alongside messages (the message *footprints*). We assume an infinite set $\mathtt{LVar} = \{x, y, \ldots\}$ of *logical variables*, distinct from program variables in $\mathtt{Var}$. Expressions in formulas are the same as program expressions (from Section 1.1), except that they are allowed to mention logical variables.

Definition 10. The grammar of logical expressions (also written E) and formulas is as follows:

$$
\begin{array}{ll}
E ::= x \mid \mathtt{x} \mid v \mid E_1 + E_2 \mid \cdots & \text{logical expressions} \\
\varphi ::= E_1 = E_2 \mid \mathsf{emp} \mid E_1 \mapsto_{\pi_1} \left(C\langle q\rangle_{\pi_2}, E_2\right) \mid E_1 \mapsto_{\pi} \left(C\langle\rangle, E_2\right) & \text{predicates} \\
\quad \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists x.\varphi \mid \varphi_1 * \varphi_2 & \text{connectives}
\end{array}
$$

The predicate $E_1 = E_2$ asserts that the two expressions E_1 and E_2 evaluate to the same value. The empty resource is denoted by the emp predicate. The connectives are those of classical logic, except $*$ which is the separating conjunction. Other classical connectives such as $\vee$, $\Rightarrow$, $\forall$ can be defined as usual.

The assertion $E_1 \mapsto_{\pi_1} (C\langle q\rangle_{\pi_2}, E_2)$ represents a heap where a single endpoint E_1 is owned with permission π_1. The endpoint follows contract C, is currently is state q of C, and its peer is E_2. Moreover, a fractional permission π_2 is assigned to q. A contract state owned with permission less than the total permission 1 cannot be updated by a program. See the example in Fig. 2 and the axioms of Section 3.3 for more details. The assertion $E_1 \mapsto_{\pi} (C\langle q\rangle, E_2)$ described in Section 2.2 (and used in Fig. 2) is simple syntactic sugar that distributes equal permissions to the endpoint and its contract state (this is introduced purely to simplify notations as only one permission has to be specified in this case):

$$E_1 \mapsto_{\pi} \left(C\langle q\rangle, E_2\right) \triangleq E_1 \mapsto_{\pi} \left(C\langle q\rangle_{\pi}, E_2\right)$$

In some cases, we will need to talk about endpoints whose contract state is not known (or owned) at all by the program, i.e. the permission on the contract state is "empty". This is denoted by assertions of the form $E_1 \mapsto_{\pi} (C\langle\rangle, E_2)$,[3] whose other parameters have the same meaning as for $E_1 \mapsto_{\pi} (C\langle q\rangle_{\pi'}, E_2)$. An empty permission on the contract state prevents all communication on that endpoint; only the knowledge that the endpoint exists is retained. As we will see in Section 3.5, we need this second form of endpoint ownership for technical reasons, to be able to describe some footprints in a way that guarantees leak freedom.

As usual, we may omit permission subscripts when they are equal to 1.

Semantics We assume additional infinite sets $\mathtt{Endpoint} = \{\varepsilon, \ldots\}$, $\mathtt{Ctt} = \{C, \ldots\}$, and $\mathtt{Control} = \{q, \ldots\}$ of respectively endpoint locations, contracts, and contract states. Remember that $\Pi = (0, 1]$ is the set of fractional permissions introduced in Section 2.2. An endpoint is associated to a tuple of one of two forms: either (ε, π, C) or $(\varepsilon, \pi, C, \pi', q)$, recording its peer endpoint ε, the fraction of the endpoint that is owned, its contract C, and possibly its current state in the contract q

[3] Note that permission models do not consider the empty permission to be a valid permission; in particular, $0 \notin \Pi$. This and the fact that specifying the contract state would be redundant when it is not known justify the need for a second notation.

and the permission π' held on that control state. Intuitively, an endpoint record (ε, π, C) corresponds to the situation in which the permission on the contract state is 0 (but 0 is not a valid permission), hence is basically unconstrained.

Endpoints are allocated in a shared heap represented by a partial function from locations to endpoints records. Remember that $\mathtt{Val}$ is the set of all values introduced in Section 1.1. The set of values $\mathtt{Val}$ contains all values of interest, for example $\mathtt{Endpoint} \cup \mathbb{N} \subseteq \mathtt{Val}$.

Definition 11 *(Local states). Local states* are pairs (s, h) of a stack $s \in \mathtt{Stack}$, mapping program variables to their values, and a heap $h \in \mathtt{Heap}$, recording the currently owned endpoints:

$$\mathtt{Stack} \triangleq \mathtt{Var} \rightharpoonup \mathtt{Val} \qquad \mathtt{Heap} \triangleq \mathtt{Endpoint} \rightharpoonup_{\mathit{fin}} \Pi \times \mathtt{Endpoint} \times \mathtt{Ctt} \times (\emptyset + \Pi \times \mathtt{Control})$$

Note that local states do not track the precise contents of the endpoint queues, only the contract state of each owned endpoint. This is enough to interpret the statements of our logic, but not for giving a faithful operational semantics to our programming language. Section 5 extends local states with queue contents and defines an operational semantics for the language.

We define the peer function $\mathit{mate}(h) : \mathtt{Endpoint} \rightharpoonup \mathtt{Endpoint}$ as the function with the same domain as h and such that $\mathit{mate}(h)(\varepsilon) = \varepsilon'$ if $h(\varepsilon) = (-, \varepsilon', -, -, -)$ or $h(-, \varepsilon, -)$. Similarly, we define the functions $\mathit{contract}(h)$ and $\mathit{cstate}(h)$ as follows: if $h(\varepsilon) = (-, -, C, -, q)$, then $\mathit{contract}(h) = C$ and $\mathit{cstate}(h) = q$, and if $h(\varepsilon) = (-, -, C)$ then $\mathit{contract}(h) = C$, the functions being undefined elsewhere.

Definition 12 *(Well-formed heap).* A heap h is said to be *well-formed* if, for all allocated ε such that $\mathit{mate}(h)(\varepsilon) = \varepsilon' \in \mathit{dom}(h)$,

- $\mathit{contract}(h)(\varepsilon)$ and $\mathit{contract}(h)(\varepsilon')$ are dual
- $\mathit{mate}(h)(\varepsilon') = \varepsilon$

We now define a partial composition operation on local states. Intuitively, the composition of two states adds together the permissions of the heap objects of these states. In particular, the composition of two states with disjoint domains is their disjoint union.

Definition 13 *(Composition of heaps).* Let h_1, h_2 be two heaps. Recall that $\oplus$ is the addition operator over real numbers, and consider the total function f defined on every endpoint location ε as:

- if $\varepsilon \notin \mathit{dom}(h_1) \cup \mathit{dom}(h_2)$, then $f(\varepsilon) = \mathit{undef}$
- if $\varepsilon \in \mathit{dom}(h_1) \setminus \mathit{dom}(h_2)$, then $f(\varepsilon) = h_1(\varepsilon)$
- if $\varepsilon \in \mathit{dom}(h_2) \setminus \mathit{dom}(h_1)$, then $f(\varepsilon) = h_2(\varepsilon)$
- if $h_1(\varepsilon) = (\pi_1, \varepsilon', C)$, $h_2(\varepsilon) = (\pi_2, \varepsilon', C)$, and $\pi_1 \oplus \pi_2 \leq 1$, then $f(\varepsilon) = (\pi_1 \oplus \pi_2, \varepsilon', C)$
- if $h_1(\varepsilon) = (\pi_1, \varepsilon', C, \pi_1', q)$, $h_2(\varepsilon) = (\pi_2, \varepsilon', C)$, and $\pi_1 \oplus \pi_2 \leq 1$, then $f(\varepsilon) = (\pi_1 \oplus \pi_2, \varepsilon', C, \pi_1', q)$
- if $h_1(\varepsilon) = (\pi_1, \varepsilon', C)$, $h_2(\varepsilon) = (\pi_2, \varepsilon', C, \pi_2', q)$, and $\pi_1 \oplus \pi_2 \leq 1$, then $f(\varepsilon) = (\pi_1 \oplus \pi_2, \varepsilon', C, \pi_2', q)$
- if $h_1(\varepsilon) = (\pi_1, \varepsilon', C, \pi_1', q)$, $h_2(\varepsilon) = (\pi_2, \varepsilon', C, \pi_2', q)$, $\pi_1 \oplus \pi_2 \leq 1$, and $\pi_1' \oplus \pi_2' \leq 1$, then $f(\varepsilon) = (\pi_1 \oplus \pi_2, \varepsilon', C, \pi_1' \oplus \pi_2', q)$
- otherwise, $f(\varepsilon) = \mathit{error}$

Let h be the heap defined by restricting f to endpoint locations ε such that $f(\varepsilon) \notin \{\mathit{undef}, \mathit{error}\}$. We say that h_1 and h_2 are *compatible*, written $h_1 \perp h_2$, if h is a well-formed heap and, for all ε, $f(\varepsilon) \neq \mathit{error}$. When this is the case, we say that h is the *composition* of h_1 and h_2, and write $h = h_1 \bullet h_2$.

In the definition above, *error* is used to represent incompatibility between two heaps at a given location, for instance because the combined permissions at that location are greater than 1, or because the heaps disagree on the peer or the contract of that location. One can check that $\bullet$ is associative and commutative with unit $\emptyset$, the function with the empty domain.

We interpret formulas using a forcing relation on well-formed local states together with an interpretation ι for logical variables:

$$\iota : \mathtt{LVar} \to \mathtt{Val}$$

Definition 14 *(Semantics of formulas).* Let $[\![E]\!]_{s,\iota}$ denote the semantics of the expression E with respect to the stack s and interpretation ι, i.e. $[\![x]\!]_{s,\iota} = s(x)$, $[\![v]\!]_{s,\iota} = v$, $[\![E_1 + E_2]\!]_{s,\iota} = [\![E_1]\!]_{s,\iota} + [\![E_2]\!]_{s,\iota}$, etc.

$$
\begin{array}{ll}
(s,h),\iota \models E_1 = E_2 & \text{iff} \quad [\![E_1]\!]_{s,\iota} = [\![E_2]\!]_{s,\iota} \\
(s,h),\iota \models \mathsf{emp} & \text{iff} \quad h = \emptyset \\
(s,h),\iota \models E_1 \mapsto_{\pi_1} \big(C\langle q\rangle_{\pi_2}, E_2\big) & \text{iff} \quad h\big([\![E_1]\!]_{s,\iota}\big) = \big(\pi_1, [\![E_2]\!]_{s,\iota}, C, \pi_2, q\big) \\
(s,h),\iota \models E_1 \mapsto_{\pi} \big(C\langle\rangle, E_2\big) & \text{iff} \quad h\big([\![E_1]\!]_{s,\iota}\big) = \big(\pi, [\![E_2]\!]_{s,\iota}, C\big) \\
(s,h),\iota \models \neg\varphi & \text{iff} \quad (s,h),\iota \not\models \varphi \\
(s,h),\iota \models \varphi_1 / \!| \varphi_2 & \text{iff} \quad (s,h),\iota \models \varphi_1 \text{ and } (s,h),\iota \models \varphi_2 \\
(s,h),\iota \models \exists x.\, \varphi & \text{iff} \quad \text{there is } v \in \mathtt{Val} \text{ such that } (s,h),[\iota \mid x : v] \models \varphi \\
(s,h),\iota \models \varphi_1 * \varphi_2 & \text{iff} \quad \text{there are } h_1, h_2 \text{ such that } h = h_1 \bullet h_2,\ (s,h_1),\iota \models \varphi_1, \text{ and } (s,h_2),\iota \models \varphi_2
\end{array}
$$

We say that φ *entails* (or *implies*) ψ, and write $\varphi \vdash \psi$ if $\varphi \Rightarrow \psi$ is *valid*, i.e. if for all well-formed local states (s,h) and all interpretations ι, $(s,h),\iota \models \varphi$ implies $(s,h),\iota \models \psi$.

Example 15. The following entailments hold.

$$
\begin{array}{rcl}
e \mapsto \big(C\langle q\rangle, f\big) * e' \mapsto \big(C'\langle q'\rangle, f'\big) & \vdash & e \neq e' \wedge f \neq f' \\
e \mapsto \big(C\langle q\rangle, f\big) * e' \mapsto \big(C'\langle q'\rangle, f'\big) & \vdash & (e = f') \Leftrightarrow (e' = f) \\
e \mapsto \big(C\langle q\rangle, f\big) * f \mapsto \big(C'\langle q'\rangle, f'\big) & \vdash & C' = \bar{C} \\
e \mapsto \big(C\langle q\rangle, f\big) & \dashv\vdash & e \Mapsto \big(C\langle q\rangle, f\big) \\
e \mapsto \big(C\langle q\rangle, f\big) & \dashv\vdash & e \Mapsto_{.5} \big(C\langle q\rangle, f\big) * e \Mapsto_{.5} \big(C\langle q\rangle, f\big) \\
e \mapsto \big(C\langle q\rangle, f\big) & \dashv\vdash & e \mapsto_{.5} \big(C\langle q\rangle, f\big) * e \mapsto_{.5} \big(C\langle\rangle, f\big)
\end{array}
$$

3.2. Footprints

Our program logic achieves *thread-modular reasoning*: each thread can be proved in isolation of other threads. There are two mechanisms at work to achieve this: first, contracts allow a thread to know what messages to expect from other threads; second, message footprints allow threads to agree on what resources are transferred alongside messages.

The *footprint* of a message is the piece of heap that is lost when sending this message and gained when receiving it.[4] The correctness of the interaction between `producer` and `consumer` in Example 6 is based on the assumption that the footprint of the `fin` message is the endpoint `f`, represented by the formula $\mathtt{f} \mapsto (C_6\langle 2\rangle, \mathtt{e})$. More generally, given a program to prove, we will associate a footprint to every message tag m used in the program and assume that every time m is sent (respectively received) the same footprint is asserted and lost (respectively assumed and added).

Definition 16 *(Footprints).* Footprints are formulas $\phi(\mathsf{src}, \mathsf{val}_1, \ldots, \mathsf{val}_n)$ where only the variables $\mathsf{src}, \mathsf{val}_1, \ldots, \mathsf{val}_n$ may appear free. The first free variable src is a parameter that stands for the endpoint that sends the message, and the next n free variables are the parameters that are to be instantiated with the message's values (assuming that it is of arity n).

Definition 17 *(Footprint environments).* A *footprint environment* Φ is a mapping from message identifiers to footprints, such that messages of arity n are mapped to footprints of arity $n+1$.

Example 18. In Example 6, we mentioned that the footprint of the `fin` message could be described by the formula $\mathtt{e} \mapsto (C_6\langle 2\rangle, \mathtt{f})$, but this is not quite accurate. Indeed, the footprint of `fin`, a message of arity zero, can have only one free variable, which represents the source endpoint. A correct footprint for this example (which allows the same proof sketch to go through) is $\phi_{\mathtt{fin}}(\mathsf{src}) \triangleq \exists f.\, \mathsf{src} \mapsto (C_6\langle 2\rangle, f)$.

Given a footprint $\phi(\mathsf{src}, \mathsf{val}_1, \ldots, \mathsf{val}_n)$, we write $\phi(E, E_1, \ldots, E_n)$ for

$$\phi[\mathsf{src}, \mathsf{val}_1, \ldots, \mathsf{val}_n < -E, E_1, \ldots, E_n]$$

3.3. Proof rules for message passing

For this section, we assume a fixed footprint environment Φ.

Channel allocation and deallocation The rules for allocation and disposal are symmetric: `open` produces two fully owned endpoints that are each other's peer, while `close` consumes them. Endpoints are allocated in the initial state of their contracts, and closing them is only valid if they are in the same state of the contract, and that state is a final state.

[4] We use a different terminology than previous work [33] where footprints were called *message invariants*.

$$\textsc{Open}\ \frac{q = \mathit{init}(C)}{\{\mathsf{emp}\}\ \mathtt{(e,f)=open(}C\mathtt{)}\ \{\mathtt{e} \mapsto (C\langle q\rangle, \mathtt{f}) * \mathtt{f} \mapsto (\overline{C}\langle q\rangle, \mathtt{e})\}} \qquad \textsc{Close}\ \frac{q \in \mathit{finals}(C)}{\{\mathtt{e} \mapsto (C\langle q\rangle, \mathtt{f}) * \mathtt{f} \mapsto (\overline{C}\langle q\rangle, \mathtt{e})\}\ \mathtt{close(e,f)}\ \{\mathsf{emp}\}}$$

Send and receive Communication instructions perform *ownership transfer* of the message footprint, and advance the control state of the communication endpoint according to the contract. More precisely, `send` first updates the control state of the endpoint, then releases ownership of the footprint corresponding to the exchanged message. Conversely, `receive` first acquires the footprint of the message, then updates the control state of the endpoint.

To account for all cases uniformly, we use a pseudo-instruction $\mathtt{skip}_{\mathtt{e}\lambda,\mathtt{f}}$, where λ denotes either $!m$ or $?m$. Its role is to update the contract state of `e` (with peer `f`) according to the action λ. The name `skip` stresses the fact that these instructions have no operational effect since the underlying semantics of programs is independent of contracts (see Section 6.2).

$$\textsc{SkipLabel1}\ \frac{q \xrightarrow{\lambda} q \in C}{\{\mathtt{e} \mapsto_\pi (C\langle q\rangle_{\pi'}, \mathtt{f})\}\ \mathtt{skip}_{\mathtt{e}\lambda,\mathtt{f}}\ \{\mathtt{e} \mapsto_\pi (C\langle q\rangle_{\pi'}, \mathtt{f})\}} \qquad \textsc{SkipLabel2}\ \frac{q \xrightarrow{\lambda} q' \in C}{\{\mathtt{e} \mapsto (C\langle q\rangle, \mathtt{f})\}\ \mathtt{skip}_{\mathtt{e}\lambda,\mathtt{f}}\ \{\mathtt{e} \mapsto (C\langle q'\rangle, \mathtt{f})\}}$$

The $\mathtt{skip}_{\mathtt{e}\lambda,\mathtt{f}}$ instruction modifies the contract state of e with respect to the action λ and checks that the transition is indeed authorised by the contract. Updating a contract state requires only a partial read permission if the state is left unchanged, and a total permission otherwise.

The pseudo-instruction $\mathtt{skip}_{\mathtt{e}\lambda,\mathtt{f}}$ is used in the rules for communications, which additionally performs ownership transfers according to the message footprint.

$$\textsc{Send}\ \frac{\{\varphi\}\ \mathtt{skip}_{\mathtt{e}!m,\mathtt{f}}\ \{\psi * \phi_m(\mathtt{e}, \overrightarrow{E})\}}{\{\varphi\}\ \mathtt{send(}m\mathtt{,e,}E_1\mathtt{,...,}E_n\mathtt{)}\ \{\psi\}} \qquad \textsc{Receive}\ \frac{\{\exists \vec{y}.\ \varphi[\overrightarrow{\mathtt{x}} \leftarrow \overrightarrow{y}] * \phi_m(\mathtt{f}[\overrightarrow{\mathtt{x}} \leftarrow \overrightarrow{y}], \overrightarrow{\mathtt{x}})\}\ \mathtt{skip}_{(\mathtt{e}?m,\mathtt{f})[\overrightarrow{\mathtt{x}} \leftarrow \overrightarrow{y}]}\ \{\psi\}}{\{\varphi\}\ \mathtt{(x_1,...,x_n) = receive(}m\mathtt{,e)}\ \{\psi\}}\ \vec{y}\ \text{fresh}$$

Note that the order in which the ownership transfer and the control state update are performed is not important unless the footprint contains the ownership of the communicating endpoint itself. Receiving assigns the message values to the variables $\mathtt{x}_1$ to $\mathtt{x}_n$ (which can also be mentioned in the footprint), hence we need to replace their previous occurrences with fresh variables y_1 to y_n.

Switch/receive Finally, two new rules address the `switch`/`receive` construct: the first rule dispatches switches on different endpoints in different subproofs, the second addresses `switch`/`receive` 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 all possible incoming messages according to the current contract state.

$$\textsc{SwitchDispatch}\ \frac{\{\varphi\}\ \mathtt{switch}\ \mathit{cases}_1\ \{\psi\} \qquad \{\varphi\}\ \mathtt{switch}\ \mathit{cases}_2\ \{\psi\}}{\{\varphi\}\ \mathtt{switch}\ \mathit{cases}_1\ \mathit{cases}_2\ \{\psi\}}$$

$$\textsc{SwitchExhaust}\ \frac{\mathit{choices}(C, q) \subseteq \{m_1, \ldots, m_n\} \qquad \{\mathtt{e} \mapsto_\pi (C\langle q\rangle_{\pi'}, \mathtt{f}) * \varphi\}\ \overrightarrow{\mathtt{x}_i}\ \mathtt{= receive(}m_i\mathtt{,e);}\ p_i\ \{\psi\}\ \text{for all } i \in \{1, \ldots, n\}}{\{\mathtt{e} \mapsto_\pi (C\langle q\rangle_{\pi'}, \mathtt{f}) * \varphi\}\ \mathtt{switch}\ \{\overrightarrow{\mathtt{x}_i}\ \mathtt{= receive(}m_i\mathtt{,e):}\ p_i\}_{i \in \{1,\ldots,n\}}\ \{\psi\}}$$

Original proof rules without sharing [33] It is interesting to recall here the first proof rules that were introduced for contract-obedient message passing by [33] (for messages of arity one only). While the specifications for opening and closing channels are similar to this work, both sending and receiving require full ownership of the communicating endpoint, which prevents fractional-shares-based sharing. Moreover, the endpoint has to be owned prior to receiving on it, contrarily to our "linearity in hindsight" principle.

$$\textsc{Send-Aplas09}\ \frac{q \xrightarrow{!m} q' \in C \qquad \mathtt{e} \mapsto (C\langle q'\rangle, \mathtt{f}) * \varphi \vdash \phi_m(\mathtt{e}, E) * \psi}{\{\mathtt{e} \mapsto (C\langle q\rangle, \mathtt{f}) * \varphi\}\ \mathtt{send(m,e,}E\mathtt{)}\ \{\psi\}}$$

$$\textsc{Receive-Aplas09}\ \frac{q \xrightarrow{?m} q' \in C}{\{\mathtt{e} \mapsto (C\langle q\rangle, \mathtt{f})\}\ \mathtt{x = receive(m,e)}\ \{(\mathtt{e} \mapsto (C\langle q'\rangle, \mathtt{f}))[\mathtt{x} \leftarrow x'] * \phi_m(\mathtt{f}[\mathtt{x} \leftarrow x'], \mathtt{x})\}}\ x'\ \text{fresh}$$

Table 1
Syntactic writes and reads of programs.

Program p	$writes(p)$	$reads(p)$
`assume(`B`)`	$\emptyset$	$fv(B)$
`x = E`	{x}	$fv(E)$
`(e,f) = open()`	{e, f}	$\emptyset$
`close(e,f)`	$\emptyset$	{e, f}
`send(a,e,`E_1`,...,`E_n`)`	$\emptyset$	{e} $\cup \bigcup_{i=1}^{n} fv(E_i)$
`(`x_1`,...,`x_n`) = receive(m,e)`	$\{x_1, \ldots, x_n\}$	{e}
`(`x_1`,...,`x_n`) = receive(m,e):` p'	$\{x_1, \ldots, x_n\} \cup writes(p')$	{e} $\cup$ $reads(p')$
`switch {` $case_1$ `...` $case_n$ `}`	$\bigcup_{i=1}^{n} writes(case_i)$	$\bigcup_{i=1}^{n} reads(case_i)$
p_1; p_2 or $p_1 \| p_2$ or $p_1 + p_2$	$writes(p_1) \cup writes(p_2)$	$reads(p_1) \cup reads(p_2)$
p'^*	$writes(p')$	$reads(p')$
`local x in` p'	$writes(p') \setminus \{x\}$	$reads(p') \setminus \{x\}$

$$\textsc{Skip}\ \frac{}{\{\mathsf{emp}\}\ \texttt{skip}\ \{\mathsf{emp}\}} \qquad \textsc{Assume}\ \frac{}{\{\mathsf{emp}\}\ \texttt{assume}(B)\ \{B \wedge \mathsf{emp}\}} \qquad \textsc{Assign}\ \frac{}{\{\varphi[\mathtt{x} \leftarrow E]\}\ \mathtt{x} = E\ \{\varphi\}}$$

$$\textsc{Sequence}\ \frac{\{\varphi\}\ p\ \{\varphi'\} \qquad \{\varphi'\}\ p'\ \{\psi\}}{\{\varphi\}\ p;\ p'\ \{\psi\}} \qquad \textsc{Parallel}\ \frac{\{\varphi\}\ p\ \{\psi\} \qquad \{\varphi'\}\ p'\ \{\psi'\}}{\{\varphi * \varphi'\}\ p \parallel p'\ \{\psi * \psi'\}}\ \dagger$$

$$\dagger\ writes(p) \cap reads(p') = reads(p) \cap writes(p') = \emptyset$$

$$\textsc{Choice}\ \frac{\{\varphi\}\ p\ \{\psi\} \qquad \{\varphi\}\ p'\ \{\psi\}}{\{\varphi\}\ p + p'\ \{\psi\}} \qquad \textsc{Star}\ \frac{\{\varphi\}\ p\ \{\varphi\}}{\{\varphi\}\ p^*\ \{\varphi\}} \qquad \textsc{Local}\ \frac{\{\varphi\}\ p[\mathtt{x} \leftarrow \mathtt{y}]\ \{\psi\}}{\{\varphi\}\ \texttt{local x in}\ p\ \{\psi\}}\ \mathtt{y} \notin fv(\varphi, p, \psi)$$

$$\textsc{Frame}\ \frac{\{\varphi\}\ p\ \{\varphi'\}}{\{\varphi * \psi\}\ p\ \{\varphi' * \psi\}}\ fv(\psi) \cap writes(p) = \emptyset \qquad \textsc{Consequence}\ \frac{\varphi' \vdash \varphi \qquad \{\varphi\}\ p\ \{\psi\} \qquad \psi \vdash \psi'}{\{\varphi'\}\ p\ \{\psi'\}}$$

$$\textsc{Conjunction}\ \frac{\{\varphi_1\}\ p\ \{\psi_1\} \qquad \{\varphi_2\}\ p\ \{\psi_2\}}{\{\varphi_1 \wedge \varphi_2\}\ p\ \{\psi_1 \wedge \psi_2\}} \qquad \textsc{Disjunction}\ \frac{\{\varphi_1\}\ p\ \{\psi_1\} \qquad \{\varphi_2\}\ p\ \{\psi_2\}}{\{\varphi_1 \vee \varphi_2\}\ p\ \{\psi_1 \vee \psi_2\}} \qquad \textsc{Existential}\ \frac{\{\varphi\}\ p\ \{\psi\}}{\{\exists x.\, \varphi\}\ p\ \{\exists x.\, \psi\}}$$

Fig. 4. Standard proof rules.

By remarking that a finer-grained treatment of the permissions required for communications is possible, we have restored the symmetry in the treatments of `send` and `receive`, and have enabled the verification of a rich variety of endpoint sharing patterns.

3.4. Proof rules of separation logic

Let us present the remaining rules of our program logic, shown in Fig. 4, which are standard in separation logic. We write $reads(p)$ (resp. $writes(p)$) for the variables read (resp. written to) by program p, as defined in Table 1. We let $fv(p)$ and $fv(\varphi)$ denote the free variables of program p and formula φ (defined as usual), respectively, and write $fv(p, \varphi)$ for $fv(p) \cup fv(\varphi)$. Let us briefly explain each rule.

- Skip: The program that does nothing does not need any resources to do so (emp). Using the frame rule, one can derive $\{\varphi\}$ `skip` $\{\varphi\}$ for any φ (using the fact that emp is the unit of $*$: $\varphi * \mathsf{emp} \Leftrightarrow \varphi$).
- Assume: If the test is successful, the program terminates.
- Assign: The precondition is updated to reflect the new value of `x` in the postcondition.
- Sequence: This is the classical Floyd–Hoare rule for composing programs sequentially: the postcondition 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 precondition into two disjoint portions that are valid respective preconditions for each of the two threads. The resulting postconditions are glued together to form the postcondition of the parallel composition.
- Choice and Star are standard.
- Local: The proof continues with a fresh variable `y`.
- 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* [29]). This also justifies giving the axioms for atomic commands in a minimalistic way.
- Consequence is the standard Floyd–Hoare rule, whose soundness follows directly from the definition of what a valid Hoare triple is. The notion of logical entailment is semantic (see Section 3.1).
- Conjunction, Disjunction, and Existential are straightforward (note that $x \notin fv(p)$ in the Existential rule, since x has to be a logical variable).

We write $\vdash_\Phi \{\varphi\}\ p\ \{\psi\}$ to denote that the Hoare triple $\{\varphi\}\ p\ \{\psi\}$ has a proof in our proof system under the footprint environment Φ, and $\vdash \{\varphi\}\ p\ \{\psi\}$ when Φ is clear from the surrounding context.

3.5. Valid footprint environments

In this section, we introduce two technical fine points about footprint environments: they must be *valid*, and the each footprint must be *precise*.

Precision is required for the *locality* of the `send` instruction (see Lemma 44 and its proof in Appendix A.2). This technicality is required in many variants of concurrent separation logic, from its inception by O'Hearn [28] to, e.g., extensions to storable locks [17]. A formula is precise if for all states, there is at most one substate satisfying it. All the footprints defined in this paper are precise.

Definition 19 *(Precision).* A formula φ is *precise* if, for all s, h, ι, there is at most one h' such that $\exists h''.\, h = h' \bullet h''$ and $(s, h'), \iota \models \varphi$.

Validity of footprint environments ensures that exchanging messages does not create memory leaks that might be missed by the logic. Although we delay the definition of validity until Section 6.3, let us describe it here informally. As remarked by Bono et al. [3] and Villard [32], if one is not careful then the program logic may miss some memory leaks. For instance, the Hoare triple

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

is derivable (for some C) using a footprint environment that assigns to the `channel` message the footprint

$$\phi_{\texttt{channel}}(s, x) \triangleq \exists y.\, s = x \wedge s \mapsto \left(C\langle q\rangle, y\right) * y \mapsto \left(\overline{C}\langle q'\rangle, s\right)$$

even though this program did not deallocate all the memory. The problem is that the ownership of the endpoint `f` is passed to the recipient of the message, but the recipient of the message is the owner of `f`, which results in a circularity. The endpoint `f` becomes "ownerless".

Several sufficient (but not necessary) conditions to prevent this situation have been explored in the literature:

- *à la* Sing♯, forbidding the message footprints to contain endpoints that are in a receive state;
- *à la* Villard [32], allowing to send server endpoints only (the first ones of the pairs allocated with `open`), and allowing only to send them from server endpoints;
- *à la* Bono et al. [3], imposing a well-foundedness condition.

We will formalise the notion of *valid footprint environments* in Section 6.3, which also forbids this situation, using a semantic criterion.

As an example of a non-valid footprint environment, consider again the proof sketch of Fig. 3. The footprint of the message `product_descr` message we implicitly used in this proof is $\exists x.x \mapsto (\overline{C_8}\langle 1\rangle, \mathsf{src})$. The footprint environment that contains this footprint is not valid, because it allows to derive a proof of for a triple of the form $\{\mathsf{emp}\}$ `p` $\{\mathsf{emp}\}$, with p leaking memory along the same lines as the example above based on the `channel` message.

In order to prove the example of Fig. 3 with a valid environment, we can change the proof slightly and rather consider that the seller loses the full ownership about the contract state of the endpoint, but not about the endpoint itself. Then the footprint for `product_descr` becomes $\exists x.\, x \mapsto_{.5} (C_8\langle 1\rangle, \mathsf{src})$. This gives rise to the following new proof:

```
seller(e) [e↦(C8⟨1⟩,f) ∗ f↦(C̄8⟨1⟩,e)] {
 local price = 0;
 while (!good(price))
 [e↦(C8⟨1⟩,f) ∗ f↦(C̄8⟨1⟩,e)] {
   send(product_descr,e);
   [e↦(C8⟨2⟩,f) ∗ f↦.5 (C8⟨⟩,e)]
   price = receive(offer,e);
 }
}

main [emp] {
 (e,f) = open();
 [(e↦(C8⟨1⟩,f) ∗ f↦(C̄8⟨1⟩,e)) ∗ emp ∗ ··· ∗ emp]
 seller(e) || buyer(f) || ... || buyer(f);
 [(e↦(C8⟨1⟩,f) ∗ f↦(C̄8⟨1⟩,e)) ∗ emp ∗ ··· ∗ emp]
 [e↦(C8⟨1⟩,f) ∗ f↦(C̄8⟨1⟩,e)]
 close(e,f);
} [emp]
```

```
buyer(f) [emp] {
 local x;
 receive(product_descr,f);
 [f↦.5 (C̄8⟨2⟩,e)]
 x = think_about_it();
 send(offer,f,x);
} [emp]
```

Our other examples so far never send the recipient endpoint along with the message, hence have valid footprint environments.

4. Further examples

4.1. Dynamic locks

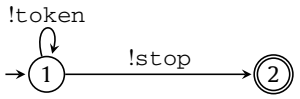
The sharing patterns we consider are expressive enough to encode dynamically allocated locks. The specifications we give to the locking primitives follow the ones of Gotsman et al. [17]. The lock intends to protect a piece of heap that satisfies a certain invariant ϕ. When the lock is acquired the ownership of ϕ is $*$-conjoined to the current state. The invariant ϕ must be established back upon releasing the lock and is then consumed, i.e. removed from the current local state of the program. Like Gotsman et al. [17], we assume that the lock is initially acquired by the thread that creates it (hence its next move should be to release it), and that it can only be released by a thread that has acquired the lock first. We propose the following encoding of locking primitives:

```
(x0,x1) = new_lock() [emp] {
  (x0,x1) = open(C);
} [locked(x1)]

dispose_lock(x0,x1) [locked(x1)] {
  send(stop,x0); receive(stop,x1);
  close(x0,x1);
} [emp]
```

```
acquire(x0,x1) [emp] {
  receive(token,x0);
} [locked(x1) ∗ φ]

release(x0,x1) [locked(x1) ∗ φ] {
  send(token,x1);
} [emp]
```

The lifetime of a lock is as follows: `new_lock` allocates a new locked lock, which can then be released with `release` and acquired again with `acquire`. Any thread can attempt to acquire the lock. The lock can be destroyed with `dispose_lock`.

The encoding above is based on two messages `token` and `stop`. The first one is used to transfer the ownership of the lock from a thread to the next thread that acquires the lock. The second one triggers the deallocation.



$$\phi_{\texttt{token}}(\mathsf{src}) \triangleq \texttt{locked}(\mathsf{src}) * \varphi$$
$$\phi_{\texttt{stop}}(\mathsf{src}) \triangleq \mathsf{emp}$$

All messages transit through the endpoints `x0` and `x1`. The ownership of the endpoints `x0` and `x1` is shared linearly using the same form of backward reasoning as explained in Section 2.3. The `token` message thus transfers the write ownership of `x0` and `x1`, and gives the right to receive the `token` message in the `acquire` function even if the endpoint `x0` is not owned in the prestate. The macro predicates are defined as follows:

$$\texttt{locked}(x_1) \triangleq \exists x.\, x_1 \mapsto \left(\overline{C}\langle 1\rangle, x\right) * x \mapsto \left(C\langle 1\rangle, x_1\right)$$

Note on memory leaks The specifications and footprints given above suffer from the memory leak issue raised in Section 3.5. This issue may be resolved by adding a *lock handle* to each lock, which is used to keep track of which threads may attempt to acquire the lock, as defined by Gotsman et al. [17]. Since multiple threads may hold the same handle, the predicate is parameterised by a fractional permission. In our encoding, the handle would be a fractional permission of the endpoints of the lock, with no permission on their contract states (similar to the example in Section 3.5). The `locked` predicate asserts partial ownership of the endpoints and full ownership of their contract states. Combining the fully-owned handle with the fact that the lock is locked gives the necessary full permission to close the channel.

$$\texttt{handle}(x_1, \pi) \triangleq \exists x.\, x_1 \mapsto_{\pi/2} \left(C\langle\rangle, x\right) * x \mapsto_{\pi/2} \left(C\langle\rangle, x_1\right)$$
$$\texttt{locked}'(x_1) \triangleq \exists x.\, x_1 \mapsto_{.5} \left(C\langle\overline{1}\rangle, x\right) * x \mapsto_{.5} \left(C\langle 1\rangle, x_1\right)$$

The specifications become:

$\{\mathsf{emp}\}$ `(x0,x1) = new_lock()` $\{\mathsf{handle}(\texttt{x1}, 1) * \mathsf{locked}'(\texttt{x1})\}$

$\{\mathsf{handle}(\texttt{x1}, 1) * \mathsf{locked}'(\texttt{x1})\}$ `dispose_lock(x0,x1)` $\{\mathsf{emp}\}$

$\{\mathsf{handle}(\texttt{x1}, \pi)\}$ `acquire(x0,x1)` $\{\mathsf{handle}(\texttt{x1}, \pi) * \mathsf{locked}'(\texttt{x1}) * \varphi\}$

$\{\mathsf{locked}'(\texttt{x1}) * \varphi\}$ `release(x0,x1)` $\{\mathsf{emp}\}$

Crucially, once the lock is created, the handle never disappears from the program logic's sight until the lock is destroyed. The ownership circularity issue that causes memory leaks is also present in the system of Gotsman et al. [17] for locks (since the invariant φ of a lock may refer to other locks), who also define a semantic criterion that prevents it. Our own criterion on footprints tackles the same issue in our encoding.

4.2. Posix-style barriers

We now give a possible encoding of synchronisation barriers using message passing and produce a proof of it. Synchronisation barriers are used to synchronise N threads in the following way: each thread calls a function `barrier_wait(x)`, and returns from this call only when all other threads have done the same.

Our encoding is based on the following idea: when the barrier is allocated, a channel is created, and a message `token` is sent on one endpoint. Later, when a client thread calls `barrier_wait`, it receives the token message and sends it again. In the meantime, it increments a counter that is passed with the message. The counter stores how many threads already passed the token. Then, every thread has to wait first for an acknowledgement message `ack`. The thread that receives the last token message, instead of passing the token once more, sends the acknowledgement, whereas other threads will receive and forward the acknowledgement exactly as they did for the token message. This ensures that all threads pass the barrier at the same time.

We write `x = new_barrier()` to indicate that `new_barrier()` returns a value that is represented by `x` in its body.

```
x = new\_barrier() [emp] {
  local y;
  (y,x) = open(C);
  send(token,y,y,0);
} [emp]

dispose\_barrier(x) [OUT^N] {
  local y,n;
  (y,n) = receive(token,x);
  close(y,x);
} [OUT^N]
```

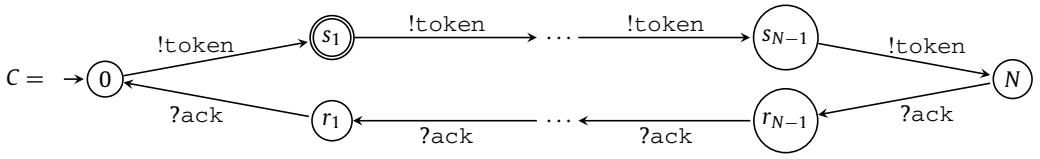
```
barrier_wait(x) [IN] {
  local y,n;
  (y,n) = receive(token,x);
  if (n = N-1)
    send(ack,x,n);
  else {
    send(token,y,y,n+1);
    n = receive(ack,y);
    if (n = 0)
      send(token,y,y,0);
    else
      send(ack,x,n-1);
  }
} [OUT]
```

The formulas IN, OUT in the specifications are the resources that are respectively lost and gained by traversing the barrier. When the last thread hits the barrier, the permissions in the N IN formulas are reshuffled into N OUT formulas, hence we require that $\mathsf{IN}^N \vdash \mathsf{OUT}^N$, writing φ^N for $\underbrace{\varphi * \cdots * \varphi}_{N \text{ times}}$.[5] When we dispose the barrier, we assume the barrier has

[5] Note that φ^N need not be inconsistent thanks to fractional permissions (e.g. φ could be a permission $1/N$ on an endpoint).

been fully used and all threads have terminated or delegated the closure of the barrier to the current thread. For this reason, we assume the client code that call the barrier disposal function owns OUT^N.

We moreover assume $\mathsf{OUT}^N * \mathsf{IN} \vdash \bot$, which suffices to ensure that the variable n in `dispose_barrier` is equal to 0 when the call to `dispose_barrier` terminates. The following contract and auxiliary specifications finish our description of the behaviour of barriers:



$$\phi_{\texttt{token}}(\mathsf{src}, y, n) \triangleq y = \mathsf{src} \wedge \mathsf{IN}^n * \exists x.\, \mathsf{src} \mapsto \left(C\langle s_{n+1}\rangle, x\right) * x \mapsto \left(\overline{C}\langle s_n\rangle, \mathsf{src}\right)$$

$$\phi_{\texttt{ack}}(\mathsf{src}, n) \triangleq \mathsf{OUT}^n * \exists y.\, y \mapsto \left(C\langle r_n\rangle, \mathsf{src}\right) * \mathsf{src} \mapsto \left(\overline{C}\langle r_{n+1}\rangle, y\right)$$

This encoding of barriers is fairly limited, as the result of the barrier synchronisation only transfers ownership of the same invariant across threads. A more powerful treatment of barriers has been explored by Hobor and Gherghina [19]. It would be interesting to encode their system (which uses a form of contracts tailored to barriers) into ours.

The encoding above again suffers from potential memory leaks: the footprint environment is invalid. A similar solution as for our encoding of locks can be developed to circumvent it.

5. Operational semantics

In this section, we give an operational semantics both for contracts and for our programming language.

5.1. Semantics of contracts

As we mentioned in Section 1.2, not all contracts are born equal: a "good" contract has to ensure the absence of unspecified receptions and orphan messages. Let us formalise these criteria by providing a semantics to channel contracts and the corresponding error configurations.

The semantics of a contract is provided by the transition system of a pair of communicating finite state machines (CFSMs), consisting of the contract and its dual. We now recall the transition systems associated to communicating finite state machines. We write Σ^* to denote the set of words over the alphabet of tags Σ, and $\sqcup$ to denote the empty word.

Definition 20 *(Configuration, initial configuration).* A *configuration* of the contract $C = (Q, \delta, q_0, F)$ is a tuple

$$\left(q, q', w, w'\right) \in Q \times Q \times \Sigma^* \times \Sigma^*$$

where q and q' are the control states of each participant in C and $\overline{C}$, and w and w' are the buffer contents in both directions. The *initial configuration* is $(\mathit{init}(C), \mathit{init}(C), \sqcup, \sqcup)$.

We now introduce a transition system that models how a configuration can evolve to another one. Informally, a configuration evolves to another one if one of the two participants (0 or 1) has triggered a transition of its contract. While triggering the transition, the contract state of this participant is updated, and if the transition is labelled with a send action (resp. a receive one), a message is added in the other's queue (resp. is popped from his own queue).

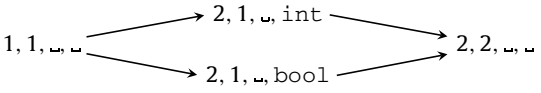
Definition 21 *(Transition relation).* Given a contract $C = (Q, \delta, q_0, F)$, we write

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

if and only if there is $i \in \{0, 1\}$, $\lambda \in \{!, ?\} \times \Sigma$ and $q_1^i \xrightarrow{\lambda} q_2^i \in C_i$ (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 contract C_2 of Example 2 is



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. Based on this observation, one can prove that most safety properties of interest are undecidable for contracts [25].

We write $\mathit{choices}(C, q)$ for the set of message identifiers m such that $\exists q'.\, q \xrightarrow{?m} q' \in C$.

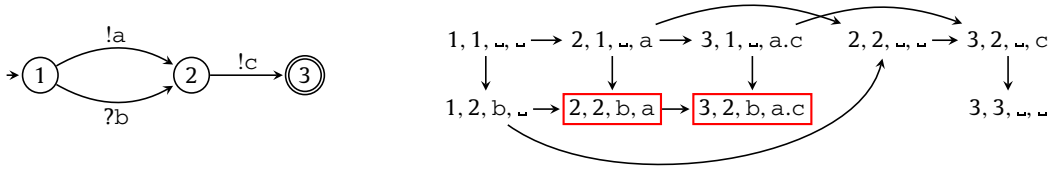
The reachable configurations of a contract C are those resulting from any number of successive transitions starting from the initial configuration.

Definition 22 *(Reachable configurations).* A configuration γ of contract C is *reachable* if $\gamma_0 \to^* \gamma$, where γ_0 denotes the initial configuration and $\to^*$ denotes the reflexive transitive closure of $\to$.

Definition 23 *(Error configurations).* Let C_0, C_1 be two contracts such that $C_1 = \overline{C_0}$. A configuration (q_0, q_1, w_0, w_1) of C_0 is called an *orphan message* if $q_0 = q_1 \in \mathit{finals}(C_0)$ and $w_0.w_1 / = \lrcorner$, and an *unspecified reception* if there is $i \in \{0, 1\}$ such that $w_i = m.w_i'$ for some w_i' and $m \notin \mathit{choices}(C_i, q_i)$.

The error configurations are the ones that correspond to communication errors, namely unspecified receptions and orphan messages.

Example 24. The figure below represents a contract and its transition system. Two of its reachable configurations are unspecified receptions. We have highlighted them with a frame.



Definition 25 *(Valid contracts).* A contract C is *valid* if all reachable configurations are neither orphan messages nor unspecified receptions.

Theorem 26. *Well-formed contracts are valid.*

Proof. See for instance Lozes and Villard [25]. □

All the contracts used in our examples are well-formed, and thus valid (except of course Example 24).

5.2. Queues and open states

Open states As already mentioned, local states are enough for interpreting formulas, but, since they abstract the content of the queues, are not suited for giving an operational semantics to our programming language. We now enrich local states with information about the content of the queues.

Definition 27 *(Queue contexts).* A *queue context* is an element $k \in \mathtt{QContext}$, where

$$\mathtt{QContext} \triangleq \mathtt{Endpoint} \to (\Sigma \times \mathtt{Val}^* \times \mathtt{Heap})^*$$

A queue context is always implicitly assumed to be *finite*, i.e. the set of ε such that $k(\varepsilon) \neq \lrcorner$ is finite.

Intuitively, a queue context k associates to every endpoint ε a queue $k(\varepsilon)$, modelled as a sequence of messages, where each message is a triple $(m, \vec{v}, h)$ consisting of a message identifier m, a tuple of values (of the same length as the arity specified by the message identifier), and a heap (the one being transferred with the message, or *owned* by the message).

Example 28. Let us define a running example. Let ε_0, ε_1, ε_0', ε_1' be four distinct endpoint locations, and h_0, h_1 the local heaps such that $h_i = \{\varepsilon_i \mapsto (1, \varepsilon_i', C, 1, q)\}$, i.e. single-endpoint heaps where the endpoint is fully owned, and is currently in the state q of contract C. Let ε, ε' be two more endpoints, distinct from the previous ones, m a message identifier with one parameter, and let k be the queue context such that $k(\varepsilon) = (m, \varepsilon_0, h_0).(m, \varepsilon_1, h_1)$ and $k(\varepsilon'') = \lrcorner$ for all $\varepsilon'' \neq \varepsilon$. This queue context models the situation where ε_0 and ε_1 have been sent on endpoint ε' with full permission and are ready to be received on ε.

Definition 29 *(Open states).* An *open state* is an element σ of

$$\texttt{OState} \triangleq \texttt{Stack} \times \texttt{Heap} \times \texttt{QContext}$$

An open state is a local state (s, h) extended with a *queue context* k that represents the content of the queues of all endpoints. The local heap h in an open state (s, h, k) represents what is currently owned by the program, while k contains pieces of heap in transit and "owned" by the relevant messages, waiting to be received and added to the local state. The semantics defined in the next section will explicitly implement ownership transfers between the local heap and the queues. Observe that our definition of open states is too permissive as it stands. For instance, it allows an endpoint to be simultaneously present in the local state and in another heap in a queue. In the rest of this section, we introduce three restrictions to our definition of open states in order to ensure that such inconsistencies are ruled out: *well-separation*, *footprint consistency*, and *contract consistency*.

Well-separation The first restriction prevents endpoints from being simultaneously owned by a message in a queue and by the local state (unless with compatible permissions).

Let $\bigodot$ denote the n-ary composition of heaps:

$$\bigodot\{h_1, \ldots, h_n\} \triangleq h_1 \bullet \cdots \bullet h_n.$$

Definition 30 *(Well-separated open states).* An open state $\sigma = (s, h, k)$ is *well-separated* if

$$LS_\sigma \triangleq h \bullet \bigodot_{\varepsilon \in dom(k)} \bigodot_{(m,v,h_m) \in k(\varepsilon)} h_m$$

is defined and well-formed.

Intuitively, an open state is well-separated if all pieces of the heap that are carried by all messages in all queues are disjoint from each others and from the local state.

Example 31. Consider the open state $\sigma = (\emptyset, h, k)$ where k is the queue context k of Example 28, and $h = \{\varepsilon_2 \mapsto (1, \varepsilon_2', C_2, 1, q_2)\}$ for some ε_2, ε_2', C_2, q_2. Then σ is well-separated if and only if ε_2 is distinct from ε_0 and ε_1 ($LS_\sigma = h \bullet h_0 \bullet h_1$).

Flattening an open state "brings at the front" all pieces of the heap that are in the queues and glues them together with the local heap. The resulting local heap is LS_σ, and the resulting queue is one where all the message footprints have been replaced with the empty heap $\emptyset$.

Let $emp(k)$ be the queue context obtained by setting all the footprints to $\emptyset$, i.e. by the lifting to k of the function $(a, v, h) \mapsto (a, v, \emptyset)$.

Definition 32 *(Flattening).* The *flattening* $flat(\sigma)$ of a well-separated open state $\sigma = (s, h, k)$ is

$$flat(\sigma) \triangleq \big(s, LS_\sigma, emp(k)\big)$$

Example 33. Recall the open state σ from Example 31. Then $flat(\sigma) = (\emptyset, h \bullet h_0 \bullet h_1, k')$, where $k'(\varepsilon) = (m, l_0, u).(m, l_1, u)$.

Footprint consistency The second restriction ensures that pieces of heap attached to messages in queues satisfy the corresponding footprints.

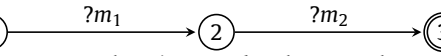
Definition 34 *(Footprint consistency).* An open state $\sigma = (s, h, k)$ is *consistent w.r.t. a footprint environment* Φ, written $\Phi \vdash \sigma$, if, for all ε, if $mate(h)(\varepsilon) = \varepsilon'$, and $k(\varepsilon) = (m_1, \vec{v}_1, h_1) \ldots (m_n, \vec{v}_n, h_n)$, then for all $i \in \{1, \ldots, n\}$,

$$h_i \models \phi_{m_i}\big(\varepsilon', \vec{v}\big)^6$$

In other words, an open state is consistent with a footprint environment Φ if in all queues, each message with tag m carrying a piece of heap h is such that h satisfies the footprint associated to m in Φ.

Example 35. Recall the state σ from Example 31 and let Φ be the footprint environment where m is assigned the footprint $\phi_m(s, x) = \exists y.\, x \mapsto (C\langle q\rangle, y)$. Then σ is consistent with respect to Φ because $h_i \models \phi_m(\varepsilon', \varepsilon_i)$ for $i \in \{0, 1\}$.

[6] Note that the interpretation of $\phi_{m_i}(\varepsilon', \vec{v})$ does not depend on the stack or the interpretation of logical variables, since footprints do not refer to variables. Thus, we are justified in writing $h_i \models \phi_{m_i}(\varepsilon', \vec{v})$ for $\forall \iota.\, (s, h_i), \iota \models \phi_{m_i}(\varepsilon', \vec{v})$.

Contract consistency The third restriction ties together the contract states of the two endpoints of a given channel and the content of their queues of incoming messages. Intuitively, we want to single out the open states that can be reached by contract obedient[7] exchanges of messages. For instance, given a state σ such that ε follows contract $C =$  according to LS_σ, if the queue of incoming messages of ε contain m_2 but not m_1, then it must be the case that ε is in state 2 (or has no permission over its control state), because m_1 must has been received on ε. Otherwise, if for instance ε is in state 1 according to LS_σ, then σ is deemed contract-inconsistent. Moreover, the control state and queue of an endpoint impose well-formedness constraints on the control state and queue of its peer ε'. In this example, the queue of incoming messages of ε' must be empty, and if ε' is also allocated in LS_σ and its control state is q, then it must be the case that $q = 3$.

The general definition is based on the CFSM semantics of contracts (see Section 5.1).

Definition 36 *(Set of configurations of an open state).* Let σ be a well-separated open state, and let $\mathit{flat}(\sigma) = (s, h, k)$. To any pair of endpoints $(\varepsilon, \varepsilon')$ such that $\mathit{mate}(h)(\varepsilon) = \varepsilon'$, we associate the set $\mathit{CONFS}(\sigma, \varepsilon, \varepsilon')$ of configurations (q, q', w, w') for which the following holds:

- $w, w' \in \Sigma^* \times \Sigma^*$ are obtained by applying the first projection $(\Sigma \times \mathtt{Val}^* \times \mathtt{Heap}) \to \Sigma$ to $k(\varepsilon)$ and $k(\varepsilon')$
- $q \in \mathtt{Control}$ (resp. q') is $\mathit{cstate}(h)(\varepsilon)$ (resp. $\mathit{cstate}(h)(\varepsilon')$) if $\varepsilon \in \mathit{dom}(h)$ (resp. $\varepsilon' \in \mathit{dom}(h)$), and arbitrary otherwise.

Thus, given an open state, we associate to every channel a set of one or more CFSM configurations such that the control states are the ones prescribed by the flattened local state (if an endpoint does not appear in the flattened local state, then its control state is unconstrained), while the queue context describes the content of the queue. In particular, if for the two endpoints of the channel, we can determine the contract state, there is exactly one configuration associated to an open state, whereas in general the set of configurations associated to an open state may contain more than one configuration.

Example 37. Let k be the queue context of Example 28, and let $h = \{\varepsilon \mapsto (1, \varepsilon', C, 1, q_0)\}$. Then $\mathit{CONFS}((\emptyset, h, k), \varepsilon, \varepsilon') = \{(q_0, q, m.m, \sqcup) : q \in \mathtt{Control}\}$.

We are now ready to define contract-consistent open states. We write $\mathit{CONFS}^{wf}(C)$ for the set of configurations of C that are reachable from its initial configuration.

Definition 38 *(Contract consistency).* An open state σ is contract-consistent if it is well-separated, $\mathit{flat}(\sigma) = (s, h, k)$, and, for all pairs of endpoints $(\varepsilon, \varepsilon')$ such that $\mathit{mate}(h)(\varepsilon) = \varepsilon'$ and $\mathit{contract}(h)(\varepsilon)$ is defined,

$$\mathit{CONFS}\big(\sigma, \varepsilon, \varepsilon'\big) \cap \mathit{CONFS}^{wf}\big(\mathit{contract}(h)(\varepsilon)\big) \neq \emptyset$$

In other words, an open state σ is contract-consistent if for every channel ruled by a contract C, the following holds for each channel $(\varepsilon, \varepsilon')$ of σ:

- either σ unambiguously defines a CFSM configuration, in which case this configuration should be reachable from the initial configuration
- or σ does not prescribe a unique CFSM configuration for ε or ε', but it is possible to fill the missing information in such a way that the open state defines a reachable CFSM configuration.

Example 39. Let $\sigma = (\emptyset, h, k)$ be as in Example 37, and let C be the contract with three states q_0, q_1, q_2, such that $q_0 \xrightarrow{?m} q_1 \xrightarrow{?m} q_2$. Then $\mathit{CONFS}^{wf}(C)$ is

$$\big\{(q_0, q_0, \sqcup, \sqcup), (q_0, q_1, m, \sqcup), (q_0, q_2, m.m, \sqcup), (q_1, q_1, \sqcup, \sqcup), (q_1, q_2, m, \sqcup), (q_2, q_2, \sqcup, \sqcup)\big\}$$

As a consequence, $\mathit{CONFS}(\sigma, \varepsilon, \varepsilon') \cap \mathit{CONFS}^{wf}(C) = \{(q_0, q_2, m.m, \sqcup)\} \neq \emptyset$, and σ is contract-consistent. Let $h' = [h \mid \varepsilon' : (1, \varepsilon, \bar{C}, 1, q_1)]$ and $\sigma' = (\emptyset, h', k)$; then $\mathit{CONFS}(\sigma', \varepsilon, \varepsilon') = \{(q_0, q_1, m.m, \sqcup)\}$ and σ' is not contract-consistent, because $\mathit{CONFS}(\sigma', \varepsilon, \varepsilon') \cap \mathit{CONFS}^{wf}(C) = \emptyset$.

We can now give the definition of *well-formed* open states.

Definition 40 *(Well-formed open state).* An open state is *well-formed w.r.t.* Φ if it is well-separated, consistent w.r.t. Φ, and contract-consistent.

[7] In session type terms, these states could be seen as "well-typed states".

When Φ is clear from context and σ is well-formed w.r.t. Φ, we simply say that σ is well-formed. From now on, we restrict our attention to well-formed open states. We define a partial composition on well-formed open states, also written $\bullet$: (s_1, h_1, k_1) and (s_2, h_2, k_2) are orthogonal if $h_1 \perp h_2$, $s_1 = s_2 = s$, $k_1 = k_2 = k$, and $\sigma = (s, h_1 \bullet h_2, k)$ is well-formed. In that case, their composition is σ. Well-formed open states equipped with this partial composition form a partial commutative monoid with multiple units, where the units are all the states of the form $(s, \emptyset, k)$ for some s and k.

5.3. Instrumented operational semantics of programs

In this section, we present our first operational semantics of programs, which is instrumented in several ways (another, *closed* semantics of programs is presented in Section 6.2 and builds on top of the instrumented one). Firstly, it keeps track of the state that is logically owned by each thread, in the form of local states. Ownership transfer happens explicitly and transfers pieces of state corresponding to message footprints to and from the queue context. Secondly, thanks to the fact that we have access to the contract of owned endpoints via the local state of a thread, the semantics detects contract violations. Finally, the semantics of a thread is independent of all possible contexts that respect the ownership hypothesis (from Section 2.1) and obey endpoint contracts. This will be crucial to establish the soundness of our program logic, which reasons about each thread in isolation from its environment.

Thus, our instrumented semantics ties precise links between the semantics of programs and that of contracts and the associated footprints, and produces error states whenever these are not obeyed. This resonates closely with the rules of our program logic, and allows us to prove a first soundness statement (Theorem 43). Yet, it does not merely follow the lead of the axioms and proof rules of the logic. In particular, parallel composition will be represented by interleaving of traces. Moreover, we will show in Sections 6.2 and 6.3 how a non-instrumented semantics can be derived from it to provide a more satisfactory soundness statement (Theorem 53).

Preliminaries The operational semantics implicitly depends on the context Φ, and is defined as a non-deterministic relation $\rightarrow$ of one of two forms:

$$\text{either } p, \sigma \rightarrow p', \sigma' \quad \text{or} \quad p, \sigma \rightarrow \textbf{error}$$

where σ is assumed to be well-formed (and we will prove in Lemma 46 that σ' is then well-formed as well). We write **error** for one of the three errors that can arise during the execution of the program: **OwnError**, **MsgError** and **ProtoError**:

OwnError indicates an *ownership error*: the program has tried to access an endpoint it does not own, or owns with not enough permissions to perform the current action.

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`/`receive` is not exhaustive.

These three error states correspond to the errors defined in Section 1.4 in the following way (note the absence of correspondence with memory leaks: these will be treated separately in Section 6.2):

$$\left.\begin{array}{r}\text{ownership errors}\\ \text{contract violations}\end{array}\right\}\ \textbf{OwnError}\ /\ \textbf{ProtoError} \qquad \left.\begin{array}{r}\text{orphan messages}\\ \text{unspecified receptions}\end{array}\right\}\ \textbf{MsgError}$$

The semantics is non-deterministic: from a given program and state, there may be several transitions in the semantics, some of them leading to error states.

Notations For conciseness purposes, and whenever possible, we describe some or all of 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 boxes. A boxed premise means that there is an additional reduction to **OwnError** from a state where the premise is either false or undefined.

Moreover, to prevent having too great a number of boxed premises, there is an implicit extra transition $p, (s, h, k) \rightarrow$ **OwnError** every time a variable not in $dom(s)$ is accessed in the premise of a transition rule.

Consider for instance one of the rules for channel closure:

$$\textsc{Close-OK}\quad \frac{s(\mathtt{e}) = \varepsilon_1 \qquad s(\mathtt{f}) = \varepsilon_2 \qquad \boxed{h(\varepsilon_1) = (1, \varepsilon_2, -, -)} \qquad \boxed{h(\varepsilon_2) = (1, \varepsilon_1, -, -)}}{\mathtt{close(e,f)}, (s, h, k) \rightarrow \mathtt{skip}, (s, h \setminus \{\varepsilon_1, \varepsilon_2\}, k)}$$

It indicates that `close(e,f)` will fault with an **OwnError** whenever `e` or `f` evaluates to an endpoint that is not fully owned in the local heap h, or whenever `e` or `f` are not each other's peer, or, implicitly, whenever `e` or `f` is not allocated on the stack s.

$$\textsc{Assume}\quad \frac{[\![B]\!]_s = \mathsf{true}}{\mathtt{assume}(B), (s,h,k) \to \mathtt{skip}, (s,h,k)} \qquad \textsc{Assign}\quad \frac{[\![E]\!]_s = v}{\mathtt{x = } E, (s,h,k) \to \mathtt{skip}, ([s \mid \mathtt{x} : v], h, k)}$$

Fig. 5. Semantics of stack commands.

$$\textsc{Open}\quad \frac{\varepsilon, \varepsilon' \in \mathsf{Endpoint} \setminus \mathit{ealloc}(s,h,k) \qquad q_0 = \mathit{init}(C)}{\mathtt{(e,f) = open}(C), (s,h,k) \to \mathtt{skip}, ([s \mid \mathtt{e} : \varepsilon, \mathtt{f} : \varepsilon'], [h \mid \varepsilon : (1, \varepsilon', C, 1, q_0), \varepsilon' : (1, \varepsilon, \overline{C}, 1, q_0)], k)}$$

$$\textsc{Close-OK}\quad \frac{s(\mathtt{e}) = \varepsilon_1 \qquad s(\mathtt{f}) = \varepsilon_2 \qquad \boxed{h(\varepsilon_1) = (1, \varepsilon_2, C, 1, q)} \qquad \boxed{h(\varepsilon_2) = (1, \varepsilon_1, C, 1, q)} \qquad q \in \mathit{finals}(C)}{\mathtt{close(e,f)}, (s,h,k) \to \mathtt{skip}, (s, h \setminus \{\varepsilon_1, \varepsilon_2\}, k)}$$

$$\textsc{Close-Orphan}\quad \frac{s(\mathtt{e}) = \varepsilon_1 \qquad s(\mathtt{f}) = \varepsilon_2 \qquad k(\varepsilon_0) \neq \text{␣ or } k(\varepsilon_1) \neq \text{␣}}{\mathtt{close(e,f)}, (s,h,k) \to \mathbf{MsgError}}$$

$$\textsc{Close-Contract}\quad \frac{s(\mathtt{e}) = \varepsilon_1 \quad s(\mathtt{f}) = \varepsilon_2 \qquad h(\varepsilon_1) = (1, \varepsilon_2, C, 1, q_1) \quad h(\varepsilon_2) = (1, \varepsilon_1, \overline{C}, 1, q_2) \qquad q_1 \neq q_2 \text{ or } q_1 \notin \mathit{finals}(C)}{\mathtt{close(e,f)}, (s,h,k) \to \mathbf{ProtoError}}$$

Fig. 6. Semantics of channel creation and destruction.

Finally, we 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. Moreover, we define the following shorthand given a state $\sigma = (s,h,k)$:

$$\mathit{valloc}(\sigma) \triangleq \mathit{dom}(s) \qquad \mathit{ealloc}(\sigma) \triangleq \mathit{dom}(h) \cup \{\varepsilon : k(\varepsilon) \neq \text{␣}\}$$
$$\mathit{cstate}(\sigma) \triangleq \mathit{cstate}(LS_\sigma) \qquad \mathit{contract}(\sigma) \triangleq \mathit{contract}(LS_\sigma)$$

We define further shorthand to describe updates to the contract state of an endpoint: if $h(\varepsilon) = (\pi, \varepsilon', C, \pi', q)$, we write $[h \mid \mathit{cstate}(\varepsilon) \leftarrow q']$ for $[h \mid \varepsilon : (\pi, \varepsilon', C, \pi', q')]$.

Stack commands The semantics of stack and heap commands is standard and independent of the heap and queue context; it is presented in Fig. 5.

Channel creation and destruction The semantics of `open` and `close`, presented in Fig. 6, takes the protocol of channels into account: `open` initialises it, and `close` raises a protocol error if the channel is closed in a non-final or unknown state of the contract (rules Close-Contract*i*). If a buffer of a closed channel is not empty, a message error is raised (rule Close-Orphan). If the endpoints given as arguments to `close` do not form a channel or do not have sufficient permission on the local heap (including for their control states), an ownership error is raised (rule Close-OK). In order to avoid creating an ill-separated state, `open` takes care not to reallocate a location already present in one of the buffers.

Note that the semantics of `close` (and that of upcoming commands below) is non-deterministic, as the premises of Close-OK and Close-Orphan may be true of the same state.

Sending and receiving messages The semantics of `send` and `receive`, presented in Fig. 7, is decomposed into an ownership transfer step and the update of the endpoint's contract state. Any of these two steps may fail: the ownership transfer may fail in the send case because a message required by the environment Φ is not available, which raises an ownership error (rule Send-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 (rule Skip-OK), or because the contract does not allow the action λ, which raises a protocol error (rule Skip-Contract). In other cases, the computation proceeds without errors, either by Send-OK followed by Skip-OK. Receive works in a similar way, but the ownership transfer occurs before the update of the contract state.

The subheap relation $\preceq$ that we use above is defined as follows: given two heaps h_1 and h_2, h_1 is a subheap of h_2, written $h_1 \preceq h_2$, if there is h such that $h_1 \bullet h = h_2$. In this case, there is a unique such h which we denote by $h_2 - h_1$.

External choice The semantics of the `switch`/`receive` construct, presented in Fig. 8, can either succeed and proceed with one of its branches (rule Switch-Select), or fail, either because an unexpected message is present at the head of one of the inspected buffers (rule Switch-Unexpected), 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 (rule Switch-Contract).

$$\textsc{Skip-OK}\quad \frac{q \xrightarrow{\lambda} q' \in C \qquad \boxed{h(\varepsilon) = (\pi, \varepsilon', C, \pi', q)} \qquad \boxed{q = q' \text{ or } \pi' = 1}}{\mathtt{skip}_{\varepsilon\lambda,\varepsilon'}, (s,h,k) \to \mathtt{skip}, (s, [h \mid cstate(\varepsilon) \leftarrow q'], k)}$$

$$\textsc{Skip-Contract}\quad \frac{\boxed{h(\varepsilon) = (\pi, \varepsilon', C, \pi', q)} \qquad \nexists q'.\ q \xrightarrow{\lambda} q' \in C}{\mathtt{skip}_{\varepsilon\lambda,\varepsilon'}, (s,h,k) \to \mathbf{ProtoError}}$$

$$\textsc{Send-OK}\quad \frac{\boxed{mate(h)(\varepsilon) = \varepsilon'} \qquad k(\varepsilon') = \alpha \qquad s(\mathtt{e}) = \varepsilon \qquad [\![\overrightarrow{E}]\!]_s = \overrightarrow{v} \qquad \mathtt{skip}_{\varepsilon!m,\varepsilon'}, (s,h,k) \to \mathtt{skip}, (s, h' \bullet h_m, k) \qquad h_m \vDash \phi_m(\varepsilon, \overrightarrow{v})}{\mathtt{send}(m, \mathtt{e}, \overrightarrow{E}), (s,h,k) \to \mathtt{skip}, (s, h', [k \mid \varepsilon' : \alpha.(m, \overrightarrow{v}, h_m)])}$$

$$\textsc{Send-Skip}\quad \frac{s(\mathtt{e}) = \varepsilon \qquad mate(h)(\varepsilon) = \varepsilon' \qquad \mathtt{skip}_{\varepsilon!m,\varepsilon'}, (s,h,k) \to \mathbf{error}}{\mathtt{send}(m, \mathtt{e}, \overrightarrow{E}), (s,h,k) \to \mathbf{error}}$$

$$\textsc{Send-Error}\quad \frac{s(\mathtt{e}) = \varepsilon \quad [\![\overrightarrow{E}]\!]_s = \overrightarrow{v} \quad mate(k)(\varepsilon) = \varepsilon' \quad \mathtt{skip}_{\varepsilon!m,\varepsilon'}, (s,h,k) \to \mathtt{skip}, (s,h',k) \quad \forall h_m \preceq h'.\ h_m \nvDash \phi_m(\varepsilon, \overrightarrow{v})}{\mathtt{send}(m, \mathtt{e}, \overrightarrow{E}), (s,h,k) \to \mathbf{OwnError}}$$

$$\textsc{Receive}\quad \frac{s(\mathtt{e}) = \varepsilon \quad k(\varepsilon) = (m, \overrightarrow{v}, h_m).\alpha \quad \boxed{mate(h \bullet h_m)(\varepsilon) = \varepsilon'} \quad \mathtt{skip}_{\varepsilon?m,\varepsilon'}, ([s \mid \overrightarrow{\mathtt{x}} : \overrightarrow{v}], h \bullet h_m, [k \mid \varepsilon : \alpha]) \to \mathtt{skip}, \sigma'}{\overrightarrow{\mathtt{x}} \ \mathtt{=}\ \mathtt{receive}(m, \mathtt{e}), (s,h,k) \to \mathtt{skip}, \sigma'}$$

$$\textsc{Receive-Error}\quad \frac{s(\mathtt{e}) = \varepsilon \quad k(\varepsilon) = (m, \overrightarrow{v}, h_m).\alpha \quad \boxed{mate(h \bullet h_m)(\varepsilon) = \varepsilon'} \quad \mathtt{skip}_{\varepsilon?m,\varepsilon'}, ([s \mid \overrightarrow{\mathtt{x}} : \overrightarrow{v}], h \bullet h_m, [k \mid \varepsilon : \alpha]) \to \mathbf{error}}{\overrightarrow{\mathtt{x}} \ \mathtt{=}\ \mathtt{receive}(m, \mathtt{e}), (s,h,k) \to \mathbf{error}}$$

Fig. 7. Semantics of communications.

$$\textsc{Switch-Select}\quad \frac{\overrightarrow{\mathtt{x}_j} \ \mathtt{=}\ \mathtt{receive}(m_j, E_j), \sigma \to \mathtt{skip}, \sigma'}{\mathtt{switch}\ \{\ \overrightarrow{\mathtt{x}_i} \ \mathtt{=}\ \mathtt{receive}(m_i, E_i)\mathtt{:}\ p_i\ \}_{i \in I}, \sigma \to p_j, \sigma'}\ j \in I$$

$$\textsc{Switch-Unexpected}\quad \frac{k(s(\mathtt{e}_j)) = (m, -, -).\alpha \qquad \forall i \in I.\, s(\mathtt{e}_i) = s(\mathtt{e}_j) \text{ implies } m_i \neq m}{\mathtt{switch}\ \{\ \overrightarrow{\mathtt{x}_i} \ \mathtt{=}\ \mathtt{receive}(m_i, \mathtt{e}_i)\mathtt{:}\ p_i\ \}_{i \in I}, (s,h,k) \to \mathbf{MsgError}}\ j \in I$$

$$\textsc{Switch-Contract}\quad \frac{\exists m \in choices(contract(h)(s(\mathtt{e}_j)), cstate(h)(s(\mathtt{e}_j))).\, \forall i \in I.\, s(\mathtt{e}_i) = s(\mathtt{e}_j) \text{ implies } m_i \neq m}{\mathtt{switch}\ \{\ \overrightarrow{\mathtt{x}_i} \ \mathtt{=}\ \mathtt{receive}(m_i, \mathtt{e}_i)\mathtt{:}\ p_i\ \}_{i \in I}, (s,h,k) \to \mathbf{ProtoError}}\ j \in I$$

Fig. 8. Semantics of external choice.

$$\textsc{Choice}\quad \frac{}{p_1 + p_2, \sigma \to p_1, \sigma} \qquad \textsc{Iteration}\quad \frac{}{p^*, \sigma \to \mathtt{skip} + (p;\ p^*), \sigma}$$

$$\textsc{Sequence-Error}\quad \frac{p_1, \sigma \to \mathbf{error}}{p_1;\ p_2, \sigma \to \mathbf{error}} \qquad \textsc{Sequence}\quad \frac{p_1, \sigma \to p_1', \sigma'}{p_1;\ p_2, \sigma \to p_1';\ p_2, \sigma'}$$

$$\textsc{Sequence-End}\quad \frac{}{\mathtt{skip};\ p_2, \sigma \to p_2, \sigma} \qquad \textsc{Parallel-Error}\quad \frac{p_1, \sigma \to \mathbf{error}}{p_1 \parallel p_2, \sigma \to \mathbf{error}}$$

$$\textsc{Parallel}\quad \frac{\boxed{\underline{norace}(p_1, p_2, \sigma)} \qquad p_1, \sigma \to p_1', \sigma'}{p_1 \parallel p_2, \sigma \to p_1' \parallel p_2, \sigma'}$$

$$\textsc{Parallel-End}\quad \frac{}{\mathtt{skip} \parallel \mathtt{skip}, \sigma \to \mathtt{skip}, \sigma} \qquad \textsc{LocalVar}\quad \frac{v \in \mathsf{Val} \qquad \mathtt{y} \notin fv(p) \cup dom(s)}{\mathtt{local\ x\ in}\ p, (s,h,k) \to p[\mathtt{x} \leftarrow \mathtt{y}], ([s \mid \mathtt{y} : v], h, k)}$$

Fig. 9. Semantics of programming constructs.

Programming constructs The semantics of the remaining programming constructs is presented in Fig. 9. We introduce the predicate $\underline{norace}(p_1, p_2, \sigma)$, false if it is impossible to partition σ into two disjoint substates on which p_1 and p_2 can safely make one step.

Definition 41 *(Race detection).* $\underline{norace}(p_1, p_2, \sigma)$ holds if and only if there exist s, h_1, h_2, k such that $\sigma = (s, h_1 \bullet h_2, k)$ and

$$\begin{aligned} & p_1, (s \setminus writes(p_2), h_1, k) \nrightarrow \mathbf{OwnError} \\ \& \ & p_2, (s \setminus writes(p_1), h_2, k) \nrightarrow \mathbf{OwnError}. \end{aligned}$$

This predicate is used to generate ownership errors in Parallel, which is otherwise described by a standard interleaving semantics. The semantics of the remaining constructs is standard.

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 where the contents of the buffers may have changed. These changes include the possibility for the environment to perform sends and receives, in accordance with their contracts, over endpoints that are not fully owned by the program, and to open and close channels not visible to the program. Transitions from the environment are written using $\dashrightarrow$.

$$\textsc{Interfere}\ \frac{(s,h,k') \text{ is well-formed}}{(s,h,k) \dashrightarrow (s,h,k')} \qquad \textsc{Interference-Step}\ \frac{\sigma \dashrightarrow \sigma'}{p,\sigma \rightsquigarrow p,\sigma'} \qquad \textsc{Program-Step}\ \frac{p,\sigma \to p',\sigma'}{p,\sigma \rightsquigarrow p',\sigma'}$$

These interferences are actually an 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, provided that the new buffers do not contradict the local state. This coarse over-approximation simplifies our setting and is enough to obtain our soundness theorems. These theorems would also hold under more refined notions of interference, by virtue of them being over-approximated by this one.

6. Properties of proved programs

In this section, we establish the soundness of our program logic with respect first to the operational semantics we just defined, and then to an "erased" version of the semantics that is independent of the artefacts of the proof (such as contracts and ownership transfers). Our first theorem, Theorem 43 in Section 6.1, ensures that proved programs do not reach the **OwnError** and **ProtoError** states, but is not strong enough on its own to establish the absence of **MsgError** states, or the absence of memory leaks. In Sections 6.2 and 6.3, we solve both of these issues by providing a *run-time* semantics, where contracts have been erased and there is no operational notion of ownership (hence no ownership transfers). Our final theorem, Theorem 53, is a more comprehensive soundness statement for programs running in a closed environment.

6.1. Soundness

Let us first define the semantic notion of *validity* of Hoare triples. We first have to overcome a discrepancy between the semantics of formulas, defined in terms of *local states*, and that of programs, defined in terms of *open states*. Formulas appearing in Hoare triples (either in the pre or the postcondition) say something about what is currently owned by the program. Since this is represented by the local state embedded in an open state, we simply interpret formulas on open states by discarding the queue context component, keeping the same stack and heap:

$$(s,h,k),\iota \models \varphi \quad \text{iff} \quad (s,h),\iota \models \varphi$$

The definitions of validity and soundness follow from a standard *fault-avoiding*, *partial correctness* interpretation of Hoare triples.

Definition 42 *(Validity).* A triple is *valid with respect to a footprint context* Φ, written $\vDash_\Phi \{\varphi\}\ p\ \{\psi\}$, if, for all well-formed open states $\sigma = (s,h,k)$ such that $fv(p) \subseteq dom(s)$ and all interpretation ι of logical variables, if $\sigma,\iota \models \varphi$ then the following properties hold:

1. $p,\sigma \not\rightsquigarrow^*$ **OwnError**
2. $p,\sigma \not\rightsquigarrow^*$ **ProtoError**
3. if $p,\sigma \rightsquigarrow^* \texttt{skip},\sigma'$, then $\sigma',\iota \models \psi$

Theorem 43 *(Soundness). If* $\vdash_\Phi \{\varphi\}\ p\ \{\psi\}$ *then* $\vDash_\Phi \{\varphi\}\ p\ \{\psi\}$.

The proof of this result follows the standard structure of proofs of soundness for separation logic (see e.g. Brookes [8]), adapted to take interferences from the environment into account. The crucial ingredients are the locality lemma, which establishes the soundness of the frame rule, and the parallel decomposition lemma, which establishes the soundness of the parallel rule.

Lemma 44 *(Locality). For all programs* p *and well-formed open states* (s,h_1,k) *and all heaps* h_2 *such that* $(s,h_1 \bullet h_2,k)$ *is defined and well-formed,*

1. *if* $p, (s, h_1 \bullet h_2, k) \rightsquigarrow$ **error** *then* $p, (s, h_1, k) \rightsquigarrow$ **error**.
2. *if* $p, (s, h_1 \bullet h_2, k) \rightsquigarrow p', \sigma'$ *then either* $p, (s, h_1, k) \rightsquigarrow$ **error** *or there exist* s', h'_1, k' *such that*
   - $\sigma' = (s', h'_1 \bullet h_2, k')$
   - $p, (s, h_1, k) \rightsquigarrow p', (s', h'_1, k')$
   - $(s', h_2, k) \dashrightarrow (s', h_2, k')$

Lemma 45 *(Parallel decomposition). For all pairs of programs* p_1, p_2 *such that* $writes(p_1) \cap reads(p_2) = reads(p_1) \cap writes(p_2) = \emptyset$, *for all states* $\sigma = (s, h_1 \bullet h_2, k)$ *and* $\sigma' = (s', h', k')$,

1. *if* $p_1 \parallel p_2, \sigma \rightsquigarrow$ **error** *then* $p_1, (s \setminus writes(p_2), h_1, k) \rightsquigarrow$ **error** *or* $p_2, (s \setminus writes(p_1), h_2, k) \rightsquigarrow$ **error**
2. *if* $p_1 \parallel p_2, \sigma \rightsquigarrow p'_1 \parallel p'_2, \sigma'$ *then* $p_1, (s \setminus writes(p_2), h_1, k) \rightsquigarrow$ **error** *or* $p_2, (s \setminus writes(p_1), h_2, k) \rightsquigarrow$ **error** *or there are* h'_1 *and* h'_2 *such that* $h' = h'_1 \bullet h'_2$ *and*
   - $p_1, (s \setminus writes(p_2), h_1, k) \rightsquigarrow p'_1, (s' \setminus writes(p_2), h'_1, k')$
   - $p_2, (s \setminus writes(p_1), h_2, k) \rightsquigarrow p'_2, (s' \setminus writes(p_1), h'_2, k')$

The interested reader may find the corresponding proofs in Appendix A.2 and Appendix A.3. The well-formedness conditions in the definition of validity complicate the proofs slightly, and require the following subject reduction lemma.

Lemma 46 *(Subject reduction). If* $p, \sigma \rightsquigarrow p', \sigma'$ *and* σ *is well-formed, then* σ' *is well-formed.*

In particular, since well-formed open states are also contract-consistent, a consequence of this lemma is that contract behaviour over-approximates channel behaviour in proved programs: the contents w_1, w_2 of two queues of a channel, when paired with the contract states q_1, q_2 of the endpoints (if these are known/owned), form a configuration (q_1, q_2, w_1, w_2) that is a reachable configuration of the transition system of the contract.

In the case of polarised contracts, we get a stronger result: in any state, at least one of the queues is empty, and the other one contains a sequence of messages that corresponds to a path between q_1 and q_2 (if the first queue is empty, otherwise between q_2 and q_1) in the contract.

6.2. Closed semantics of programs

The operational semantics presented in Section 5.3 above, as noted in the beginning of this section, is instrumented to represent more directly the artefacts of the program logic: contracts and ownership transfers are given an operational meaning, whereas in a more realistic semantics they should be "in the eye of the prover" only. In this section, we define an erased semantics that is independent of these artefacts.

We model run-time executions by using the operational semantics of Section 5.3 with empty footprints and universal contracts (those with a single state that is both initial and final, and from which all transitions are allowed in the form of self-loops), without interferences from the environment,[8] and without checks for contract obedience.

Formally, we write Φ_{emp} for the proof environment in which all message footprints are replaced by emp. Given an alphabet Σ, the *universal contract* $\mathfrak{C}_\Sigma$ is the contract $(\Sigma, \lambda q.\lambda(d, m).q, q, \{q\})$. Given a program p, $\mathfrak{C}(p)$ is p where every contract (those mentioned in `open` commands) is replaced by $\mathfrak{C}_\Sigma$. Likewise, given an open state σ, $\mathfrak{C}(\sigma)$ is σ where every contract of every heap is replaced by $\mathfrak{C}_\Sigma$.

Definition 47 *(Run-time semantics).* The run-time semantics is defined as a relation $\Rightarrow$ between a program, an open state, and either another program and open state or an error, as such:

- $p, \sigma \Rightarrow p', \sigma'$ if $\mathfrak{C}(p), \mathfrak{C}(flat(\sigma)) \rightarrow p', \sigma'$ in the operation semantics instrumented by Φ_{emp};
- $p, \sigma \Rightarrow$ **error** if $\mathfrak{C}(p), \mathfrak{C}(flat(\sigma)) \rightarrow$ **error** in the operational semantics instrumented by Φ_{emp}, and if **error** $\neq$ **ProtoError**.

Clearly, the run-time semantics of a program depends neither on the footprint context, nor on the contracts associated to channels. In this sense, it is "un-instrumented". Notice that replacing contracts with universal ones suppresses all potential protocol errors in `send` and `close` commands, but makes `switch`/`receive` fault more often (since they have to accommodate all possible tags each time!). This is why the run-time semantics ignores errors raised by not following the contracts. Finally, notice also that when $p, \sigma \Rightarrow p', \sigma'$, then p' only mentions universal contracts and σ' is flat ($flat(\sigma') = \sigma' = \mathfrak{C}(flat(\sigma'))$).

We now have enough material to formally characterise the absence of memory leaks.

[8] 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.

Definition 48 *(Leak-free state).* An open state σ is *leak-free* if there is a program p such that $p, \sigma \Rightarrow^* \texttt{skip}, \sigma'$ for some σ', and $p, \sigma \Rightarrow^* \texttt{skip}, \sigma'$ implies that $\sigma' = (s', \emptyset, k)$ with k empty everywhere: for all ε, $k(\varepsilon) = \lrcorner$.

Definition 49 *(Safety).* A program p is *safe* if, for all $\sigma \models \mathsf{emp}$,

1. $p, \sigma \not\Rightarrow^*$ **error**
2. if $p, \sigma \Rightarrow^* \texttt{skip}, \sigma'$ then σ' is leak-free

6.3. Proving programs safe

Let us now show that proved programs satisfy the safety definition above. First, one has to be careful to use valid contracts only. Indeed, soundness ensures the absence of **ProtoError**, but not of **MsgError**. When all the contracts are valid, however, subject reduction guarantees that a channel will only get into reachable configurations of the contract transition system, thus the channel never gets into an unspecified reception configuration nor an orphan message configuration if the contract never does either, and thus cannot reach **MsgError**.

This is however not enough to prevent memory leaks, as mentioned in Section 3.5. Indeed, there are two ways of creating ownerless resources (those that disappear out of sight of the program logic):

- when a channel is closed, if some messages with non-empty footprints are still in the queue
- when a message is sent, if the ownership of the reception endpoint is granted by the footprint of the message

Moreover, one may make ownerless resources ownerful again by receiving a message that provides ownership of the reception endpoint a posteriori.

As an aside, ownerless resources are also undesirable in Sing♯, even though this language is garbage collected. Indeed, execution units in Sing♯ 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. Sing♯ prevents ownerless cells by imposing a condition that is quite similar to the locality condition in the pi-calculus [27]: endpoints can only be transferred if they are in a send state of the contract (Sing♯ contracts are thus necessarily polarised).

Let us first introduce the notion of *self-contained states*, which are those states that "see" all allocated resources.

Definition 50 *(Self-contained state).* A well-separated open state $\sigma = (s, h, k)$ is *self-contained* if

1. for all ε such that $k(\varepsilon) \neq \lrcorner$, $\varepsilon \in dom(LS_\sigma)$
2. for all ε such that $\varepsilon \in dom(LS_\sigma)$, $LS_\sigma(\varepsilon) = (1, -, -, 1, -)$ and $mate(\varepsilon, LS_\sigma) \in dom(LS_\sigma)$

We are now ready to formally define the notion of valid environments mentioned in Section 3.5.

Definition 51 *(Valid environment).* An environment Φ is *valid* if all contracts in Φ are valid, and for all stacks s and queue contexts k, if $(s, \emptyset, k)$ is well-formed w.r.t. Φ and self-contained then all queues are empty (for all ε, $k(\varepsilon) = \lrcorner$).

Intuitively, a footprint environment 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 52 *(Provable safety).* A program p is *provably safe* if there is a program p' that terminates from all states and a valid environment Φ such that $\vdash_\Phi \{\mathsf{emp}\}\ p; p'\ \{\mathsf{emp}\}$.

Theorem 53 *(Safety). For all program p, if p is provably safe then p is safe.*

We refer the interested reader to Appendix A.4 for the details of the proof.

7. Related works

7.1. Copyless message massing

Bono, Messa and Padovani [3] introduced a type system *à la* session type for CoreSing♯, a model of Sing♯. Their work is an alternative formalisation of copyless message passing in the sense of Villard et al. [33] recast as a type system. They accurately pointed out that previous work [33] did not prevent memory leaks, and they introduced a well-foundedness condition on footprints to fix the problem. More recently, they extended their type system with polymorphism and qualified types [4]. Their approach does not however deal with endpoint sharing.

7.2. Sharing and multirole sessions

Giunti and Vasconcelos were the first to consider the problem of sharing contract obedient endpoints. They extended session types for that purpose and introduced a distinction between linear and unrestricted channels [16]. They illustrated their approach on a "petition" protocol, where an organiser sends a petition request to participants, and then collects their signatures. In the syntax used in the present paper, this protocol is modelled by the code snippet below.

```
organiser(channels,N) {
  local e,f,i=0;
  (e,f) = open(C);
  while (i<N) { send(req,channels.(i++),f) }
  i = 0;
  while (i<N) { receive(sign,e); /* ... */ i++; }
}
```

```
signer(ep) {
  local f;
  f = receive(req,ep);
  /* ... */
  send(sign,f);
}
```

The organiser allocates a pair of endpoints pointed to by `e` and `f` for the petition, forming a bidirectional, asynchronous channel where `e` can be used to send on `f` and vice-versa. All participants are given access to the endpoint `f` in the first loop; the second loop collects all signatures on `e`. On the other side, each participant i receives a reference to `f` on its local endpoint `ep` (assumed to be paired with the endpoint `channels.i`), and uses it to send back its signature. Due to the non-linear usage of endpoint `f`, standard session types (as well as Sing♯ contracts) cannot specify such a protocol. The linear qualifier applies to endpoints that have to be used linearly, as in ordinary session types, while the unrestricted one allows any behaviour on the endpoint. The petition protocol admits the unrestrictedly qualified type $C = un?\texttt{sign}\,;\, C$. This type C guarantees that the session on channel `(e, f)` contains `sign` messages only, and it allows endpoints typed with C to be shared. To retain soundness, unrestrictedly typed channels are limited to "single state" protocols in their work [16], and they do not support "dynamically changing qualifications" as we can do with permissions. Giunti recently implemented a type-checker for qualified types [15].

Multirole session types [10] address the problem of sharing channels quite differently. The multirole session type $\forall x : \mathsf{client.server} \to x\langle m\rangle$ describes a server multicasting a message m to its clients. Roles, like client in this example, describe groups of processes, and quantification is used to replicate a communication pattern that must be followed by each process of a given role. This kind of type assumes that some advanced operations are available to the programmers, like joining a session, leaving it, or polling a role. Our work does not deal with such primitives for session management. It might seem natural to associate the type

$$\mu\mathcal{C}.\exists x : \mathsf{buyer}.\ \mathsf{seller} \to x\langle \mathsf{product\_description}\rangle; x \to \mathsf{seller}\langle \mathsf{offer}\rangle; \mathcal{C}$$

to our auction example (see Example 8). Deniélou and Yoshida however raise some issues with including such a form of existential quantification over roles in their system, and for this reason we believe that our system is incomparable with multirole session types.

7.3. Confluence and completeness

Francalanza, Rathke and Sassone [14] introduced a separation logic for a process algebra close to CCS. Their separation logic ensures that provable processes are confluent, due to a form of linearity in the usage of channels. The program logic we introduced in Villard et al. [33] ensures a similar form of linearity, and it might be asked whether the extension we introduced in this work is needed for proving non-confluent programs (like our encoding of locks). Surprisingly, the answer is that the original version of our logic already allows to prove programs that are not confluent. Consider for instance the following program.

```
p() {
  (e,f) = open();
  close(e,f);
  p1() || p2();
}
```

```
p1(){
  (e1,f1) = open();
  while (e1 != e) {
    close(e1,f1);
    (e1,f1) = open();
}
}
```

```
p2(){
  (e2,f2) = open();
  while (e2 != e) {
    close(e2,f2);
    (e2,f2) = open();
}
}
```

This program is not confluent: indeed, either both threads `p1` and `p2` diverge, or one of them manages to "recycle" the address `e` and by allocating an endpoint at `e`, but then it terminates without deallocating it and the other thread diverges. The choice of which thread terminates is non-deterministic and depends on the memory allocator and the scheduler.

Although this program is not confluent, it is provable: the Hoare triple

$$\mathsf{emp}\ \texttt{p()}\ \left\{\left(\texttt{e}\mapsto\left(C\langle 1\rangle, \texttt{f}\right) * \texttt{f}\mapsto\left(C\langle 1\rangle, \texttt{e}\right)\right) * \left(\texttt{e}\mapsto\left(C\langle 1\rangle, \texttt{f}\right) * \mapsto \texttt{f}\mapsto\left(C\langle 1\rangle, \texttt{e}\right)\right)\right\}$$

also equivalent to {emp} `p() {`$\bot$}, is easily derivable in our program logic.

With slight variations, it is also possible to derive an example that shows that our program logic is incomplete, namely that some safe programs cannot be proved. Consider the same program as above, where the threads `p1` and `p2` are extended by a last instruction `x=0`. Then we cannot prove the program, because the parallel rule disallows having the same global variable syntactically appearing as a modified variable in two parallel threads. There is however no race in this program, since at most one thread reaches the instruction `x=0`. As a conclusion, this new program would be safe, but not provable. Note that the incompleteness of concurrent separation logic due to this "trick of memory allocation" is a folklore result, and providing a proof of completeness for a concurrent separation logic is, to date, an open problem [9]. It can also be observed that all of these complications come from the fact that we have to deal with a shared memory. Were we to consider a more abstract allocation mechanism, such as a fresh name generator, these problems might become more easily solvable.

7.4. Progress

As already mentioned, our program logic does not prevent programs from having deadlocks. It is not very difficult to figure out that the following scenario is deadlocked but provable: two processes that try to exchange two messages, on two different channels, by first waiting for the message of the other process before sending its own message. The problem here resides in the fact that the two channels are ruled by different contracts, and the deadlock can only be ruled out if there is a global discipline over all channels, and more generally over all synchronisation primitives. Existing literature provides several methods for avoiding deadlocks: Kobayashi's type system is one example that applies to the synchronous π-calculus [23]. Based on global session descriptions, Bettini et al. [2] developed a framework where it is possible to establish global progress for multi-channel protocols. More recently, Leino et al. introduced a program logic [24] that ensures deadlock-freedom for programs that manipulate channels and locks. We conjecture that these mechanisms could be added our program logic in order to prevent deadlocks.

7.5. Miscellaneous

Turon and Wand [31] proposed a program logic for the (untyped) π-calculus. Their program logic, focused on temporal reasoning and refinement, also allows to share channels with fractional permissions. However, no communication contracts are supported, thus avoiding the issue of reconciling contract obedience with sharing.

Merro showed that the full π-calculus can be encoded in the local π-calculus [27]. On the contrary, the contract-obedient π-calculus underlying our model of Sing♯ seems strictly more expressive than the local, contract-obedient one. However, we did not try to characterise the expressive power of our unrestrictedly linear π-calculus.

8. Conclusion

We have developed a program logic that achieves local reasoning for message-passing programs. The program logic can be seen as a marriage of ideas from separation logic (including fractional permissions) and session types (via our communication contracts). We have introduced two novel mechanisms to reason about endpoint sharing while preserving contract obedience. One is based on an adaptation of fractional permissions to message passing programs, while the other allows threads to share endpoints between intervals of illusion of total ownership, within which a thread is free to use the endpoint as though it was not shared. We gave evidence of the expressivity of our proof system on several examples. We established its soundness via the introduction of a rather detailed operational semantics for our programming language.

The operational semantics is rather verbose with respect to details that are sometimes abstracted away in more process-algebraic presentations (such as the stack, or memory allocation). Its instrumented nature makes the definition of the semantics perhaps too subtle, and the proofs tedious. Clearly, better semantic tools are needed to reason about ownership-aware concurrent programs.

Appendix A. Proofs

A.1. Auxiliary lemmas

Lemma 54. *For all p, p', σ, σ', if $p, \sigma \rightsquigarrow p', \sigma'$ and σ is well-separated, then σ' is well-separated.*

Proof. The only case that requires special attention is channel allocation, where the new endpoints are chosen "fresh" with respect to the flattening of σ, and not just to the local state, thus preserving well-separation. □

Lemma 55. *For all footprint environments Φ, for all σ, σ', if $p, \sigma \rightsquigarrow p', \sigma'$ and $\Phi \vdash \sigma$, then $\Phi \vdash \sigma'$.*

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

Lemma 56. *For all p, p', σ, σ', if $p, \sigma \rightsquigarrow p', \sigma'$ and σ is contract-consistent, then σ' is contract-consistent.*

Proof. Straightforward. □

We get Lemma 46 as an immediate corollary of the above three lemmas.

Lemma 57. *For all σ_1, σ_2, if $\sigma_1 \bullet \sigma_2$ is well-separated, then σ_1 is well-separated.*

Proof. Straightforward by definition of well-separated. □

Lemma 58. *For all σ_1, σ_2, if $\sigma_1 \bullet \sigma_2$ is contract-consistent then σ_1 is contract-consistent.*

Proof. Let ε and ε' be some peer endpoints in $\sigma_1 \bullet \sigma_2$. By hypothesis and definition of contract consistency (Definition 38),

$$\mathit{CONFS}\big(\sigma_1 \bullet \sigma_2, \varepsilon, \varepsilon'\big) \cap \mathit{CONFS}^{wf}(C) \neq \emptyset.$$

By definition of $\mathit{CONFS}(\sigma, \varepsilon, \varepsilon')$,

$$\mathit{CONFS}\big(\sigma_1 \bullet \sigma_2, \varepsilon, \varepsilon'\big) \subseteq \mathit{CONFS}\big(\sigma_1, \varepsilon, \varepsilon'\big).$$

Thus $\mathit{CONFS}(\sigma_1, \varepsilon, \varepsilon') \cap \mathit{CONFS}^{wf}(C) \neq \emptyset$. Since these hold for any two such endpoints $\varepsilon, \varepsilon'$, σ_1 is contract-consistent (by Definition 38). □

Lemma 59. *For all σ_1, σ_2, if $\sigma_1 \bullet \sigma_2$ is well-formed then σ_1 is well-formed.*

Proof. Straightforward by Definition 40 and the previous two lemmas. □

A.2. Proof of the locality lemma (Lemma 44)

Proof. The first part of the lemma, sometimes called "safety monotonicity", is straightforward by induction on the derivation tree of $p, (s, h_1 \bullet h_2, k) \rightsquigarrow \mathbf{error}$. Let us prove the second part by induction on the derivation tree of $p, (s, h_1 \bullet h_2, k) \rightsquigarrow p', \sigma'$.

Assume first that the step was a program transition, i.e. $p, (s, h_1 \bullet h_2, k) \to p', \sigma'$, and that $p, (s, h_1, k) \not\rightsquigarrow \mathbf{error}$. The key observation is that changes between $h_1 \bullet h_2$ and h' can only concern 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 footprints. From this observation, we have $h' = h'_1 \bullet h_2$ for some h'_1 and $p, (s, h_1, k) \rightsquigarrow p', (s', h'_1, k')$. Moreover, since $(s, h_1 \bullet h_2, k)$ is contract-consistent, so is σ' by subject reduction, and hence so is (s', h_2, k') by Lemma 59. Thus, $(s', h_2, k) \dashrightarrow (s', h_2, k')$, which ends the proof for this case.

Assume now that $p, \sigma \rightsquigarrow p, \sigma'$ by $\sigma = (s, h_1 \bullet h_2, k) \dashrightarrow \sigma'$. Then $\sigma' = (s, h_1 \bullet h_2, k')$ for some k' such that σ' is well-formed. Choosing $h'_1 = h_1$ ends the proof by Lemma 59. □

A.3. Proof of the parallel decomposition lemma (Lemma 45)

Proof. The first point is a direct consequence of the rules for error propagation and the locality lemma. Let us prove the second point. Assume p_1, p_2, $\sigma = (s, h_1 \bullet h_2, k)$, and $\sigma' = (s', h', k')$ as in the statement of the theorem, and assume moreover that $p_1, (s \setminus \mathit{writes}(p_2), h_1, k) \not\rightsquigarrow \mathbf{error}$ and $p_2, (s \setminus \mathit{writes}(p_1), h_2, k) \not\rightsquigarrow \mathbf{error}$. Let us reason by case analysis on the first rule applied in the derivation tree of $p_1 \parallel p_2, \sigma \rightsquigarrow p'_1 \parallel p'_2, \sigma'$. There are only two possible cases:

- The first rule applied is the interleaving rule

$$\frac{p_1, \sigma \to p'_1, \sigma'}{p_1 \parallel p_2, \sigma \to p'_1 \parallel p_2, \sigma'}$$

  Then by the locality lemma, there is h'_1 such that $\sigma' = (s', h'_1 \bullet h_2, k')$, $p_1, (s, h_1, k) \to p'_1, (s', h'_1, k')$, and $(s, h_2, k) \dashrightarrow (s, h_2, k')$. Observe moreover that p_1 cannot modify variables in $\mathit{writes}(p_2)$, otherwise $\mathit{norace}(p_1, p_2, \sigma)$ would not hold which would violate our assumption that $p_1, (s \setminus \mathit{writes}(p_2), h_1, k) \not\rightsquigarrow \mathbf{error}$. Thus, $p_1, (s \setminus \mathit{writes}(p_2), h_1, k) \to p'_1, (s' \setminus \mathit{writes}(p_2), h'_1, k')$, and moreover $s \setminus \mathit{writes}(p_1) = s' \setminus \mathit{writes}(p_1)$. Since well-formedness does not depend on the stack, we also have that $(s \setminus \mathit{writes}(p_1), h_2, k) \dashrightarrow (s' \setminus \mathit{writes}(p_1), h_2, k')$, hence $p_2, (s \setminus \mathit{writes}(p_1), h_2, k) \rightsquigarrow p'_2, (s' \setminus \mathit{writes}(p_1), h_2, k')$, as a direct consequence of $(s, h_2, k) \dashrightarrow (s, h_2, k')$.
- The first rule applied is the interference rule, hence $s' = s$, $h' = h_1 \bullet h_2$, and $(s, h_1 \bullet h_2, k')$ is well-formed. By Lemma 59, (s, h_1, k') and (s, h_2, k') are also well-formed. Since these facts are independent of the stack s, we have $p_1, (s \setminus \mathit{writes}(p_2), h_1, k) \rightsquigarrow p_1, (s \setminus \mathit{writes}(p_2), h_1, k')$ and similarly for p_2 as required. □

A.4. Proof of the safety theorem (Theorem 53)

Proving that provable safe programs are safe is done in three steps. First, we link **ProtoError** to **MsgError** using contracts. Second, we link the run-time semantics to the proof-dependent operational semantics. Third, we link the absence of memory leaks in the proof to the same property for the run-time semantics.

The first point proceeds directly from the subject reduction lemma.

Lemma 60 *(Message safety). For all programs p, for all states σ, if the following holds*

- *σ is well-formed*
- *$p, \sigma \not\rightsquigarrow^*$ **ProtoError***
- *$p, \sigma \not\rightsquigarrow^*$ **OwnError***

*then $p, \sigma \not\rightsquigarrow^*$ **MsgError**.*

Proof. Let us assume the hypothesis, and assume, by contradiction, that $p, \sigma \rightsquigarrow^*$ **MsgError**.

There are thus p' and σ' such that: $p, \sigma \rightsquigarrow^* p', \sigma'$ and $p', \sigma'' \to$ **MsgError**.

By definition of $\to$, there must be some faulty configuration (q, q', w, w') in $\mathit{CONFS}(\sigma', \varepsilon, \varepsilon')$ for some endpoints $\varepsilon, \varepsilon'$. By subject reduction, σ' is well-formed, and thus the same set of configurations $\mathit{CONFS}(\sigma', \varepsilon, \varepsilon')$ also contains a reachable configuration of the form (q_1, q_1', w, w'). Since the contract is valid, the latter configuration cannot be faulty and hence is distinct from (q, q', w, w'). It suffices to show that $\mathit{CONFS}(\sigma', \varepsilon, \varepsilon')$ contains exactly one configuration to derive a contradiction:

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

The connection between the run-time semantics and the proof-based semantics uses flattening of open states (Definition 32).

Lemma 61 *(Runtime soundness). For all p, σ, σ',*

1. *if $p, \sigma \Rightarrow$ **error**, then $p, \sigma \rightsquigarrow$ **error**;*
2. *if $p, \sigma \Rightarrow p', \sigma'$, then*
   - *either $p, \sigma \rightsquigarrow$ **error**;*
   - *or $p, \sigma \rightsquigarrow^* p'', \sigma''$, for some σ'' such that $\mathfrak{C}(p'') = p'$ and $\mathfrak{C}(\mathit{flat}(\sigma'')) = \sigma'$.*

By straightforward induction, this lemma states that any error in the run-time semantics can be lifted to an error in the instrumented semantics: if $p, \sigma \Rightarrow^*$ **error**, then $p, \sigma \rightsquigarrow^*$ **error**.

Proof. Assume $\mathfrak{C}(\mathit{flat}(\sigma)) = (s, h, k)$ and $\sigma = (s, h_1, k_1)$. We prove each point separately. For the first point, assume that $p, \sigma \Rightarrow$ **error**; one of the following cases holds:

- an **OwnError** is triggered because $\mathfrak{C}(p), \mathfrak{C}(\mathit{flat}(\sigma)) \to$ **OwnError**: then we also have $p, \sigma \rightsquigarrow$ **OwnError** thanks to safety monotonicity (observe that $h_1 \preceq h$);
- an **MsgError** is triggered because $\mathfrak{C}(p), \mathfrak{C}(\mathit{flat}(\sigma)) \to$ **MsgError**: the error depends only on the first message identifier in the queue causing the error, which is the same in σ and $\mathfrak{C}(\mathit{flat}(\sigma))$, hence $p, \sigma \rightsquigarrow$ **MsgError**;
- a **ProtoError** is triggered: this is never the case, due to Definition 47.

For the second point, assume that $p, \sigma \Rightarrow p', \sigma'$; one of the following cases holds:

- A channel instruction is executed that transfers ownership: $\mathfrak{C}(p), \mathfrak{C}(\mathit{flat}(\sigma)) \to p', \sigma'$ under $\mathit{emp}(\Phi)$, and if $p, \sigma \not\rightsquigarrow$ **error**, $p, \sigma \rightsquigarrow p'', \sigma''$ under Φ. Then the footprint lost from (resp. added to) h_1 during that step will go inside (resp. be taken from) the queue context, hence $\mathfrak{C}(\mathit{flat}(\sigma'')) = \sigma'$.
- Any other operational rule is triggered: the semantics match. □

Finally, the following lemma will be the cornerstone of the proof that proved programs do not leak memory that is not visible to the program logic.

Lemma 62 *(Self-containment preservation). Let p, p', σ, σ' be such that*

- $p, \sigma \to p', \sigma'$
- $p, \sigma \not\to$ **error**
- σ *is well-formed and self-contained.*

Then σ' is self-contained.

Proof. Let EP_0 and EP'_0 be the sets of endpoints with permission 1 in the local heaps of LS_σ and $LS_{\sigma'}$, EP_1 and EP'_1 the domains of the local heaps of LS_σ and $LS_{\sigma'}$, and EP_2 and EP'_2 the sets of endpoints whose incoming queues are not empty in σ and σ', respectively. We have $EP_2 \subseteq EP_1 \subseteq EP_0$. Moreover, if EP_1 contains an endpoint, it also contains its peer (according to σ). We require to show that $EP'_2 \subseteq EP'_1 \subseteq EP'_0$, and that EP'_1 also contains the peer of every endpoint in EP'_1. 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, $EP'_0 = EP_0 \uplus \{\varepsilon, \varepsilon'\}$, $EP'_1 = EP_1 \uplus \{\varepsilon, \varepsilon'\}$, and $EP'_2 = EP_2$.
- for Close-OK, $EP_0 = EP'_0 \uplus \{\varepsilon, \varepsilon'\}$, $EP_1 = EP'_1 \uplus \{\varepsilon, \varepsilon'\}$, and $EP_2 = EP'_2$. Moreover, by $p, \sigma, \not\to$ **MsgError**, $\{\varepsilon, \varepsilon'\} \cap EP_2 = \emptyset$.
- for Send-OK where ε sends a message to ε', $EP'_0 = EP_0$, $EP'_1 = EP_1$, and $EP'_2 = EP_2 \cup \{\varepsilon'\}$. By $p, \sigma, \not\to$ **OwnError**, we have $\varepsilon \in EP_1$, so $\varepsilon' \in EP_1$.
- for Receive-OK where ε is receiving a message, $EP'_0 = EP_0$, $EP'_1 = EP_1$, and $EP_2 = EP'_2 \cup \{\varepsilon\}$. □

We are now ready to prove Theorem 53.

Proof. Let σ_{emp} denote an open state with no endpoint allocated and empty queues. Let Φ be a valid environment and p' be a terminating program such that $\vdash_\Phi \{\mathsf{emp}\}\ p; p'\ \{\mathsf{emp}\}$. We require to show that

1. $p, \sigma_{emp} \not\to^* \mathbf{error}$
2. if $p, \sigma_{emp} \Rightarrow^* \mathtt{skip}, \sigma$, then σ is leak-free

Let us start by proving point 1. By Theorem 43, $p; p', \sigma_{emp} \not\to^*$ **OwnError**, and $p; p', \sigma_{emp} \not\to^*$ **ProtoError**. By Lemma 60, $p; p', \sigma_{emp} \not\to^*$ **MsgError**, thus by definition of Sequence and Sequence-Error, $p, \sigma_{emp} \not\to^*$ **error**. Finally, by a straightforward induction using Lemma 61, $p, \sigma_{emp} \not\to^*$ **error**.

Let us now prove point 2. Let σ be such that $p, \sigma_{emp} \Rightarrow^* \mathtt{skip}, \sigma$. Since p' terminates, there is σ' such that $p', \sigma \Rightarrow^* \mathtt{skip}, \sigma'$ and $\sigma' \models \mathsf{emp}$. Thus, by Theorem 43, there are s' and k' such that $\sigma' = (s', \emptyset, k')$. By Lemma 46 and a straightforward induction on the number of steps using Lemma 62, σ' is well-formed and self-contained. Since Φ is valid (Definition 51), all queues are empty in k'. This shows that σ is leak-free. □

References

[1] Android Open Source Project, fetched Sept. 2013, Android developers, http://developer.android.com/.
[2] L. Bettini, M. Coppo, L. D'Antoni, M.D. Luca, M. Dezani-Ciancaglini, N. Yoshida, Global progress in dynamically interleaved multiparty sessions, in: F. van Breugel, M. Chechik (Eds.), CONCUR, Springer, 2008, pp. 418–433.
[3] V. Bono, C. Messa, L. Padovani, Typing copyless message passing, in: G. Barthe (Ed.), ESOP, Springer, 2011, pp. 57–76.
[4] V. Bono, L. Padovani, Typing copyless message passing, Log. Methods Comput. Sci. 8 (2012).
[5] R. Bornat, C. Calcagno, P.W. O'Hearn, M.J. Parkinson, Permission accounting in separation logic, in: POPL, 2005, pp. 259–270.
[6] J. Boyland, Checking interference with fractional permissions, in: SAS, 2003, pp. 55–72.
[7] D. Brand, P. Zafiropulo, On communicating finite-state machines, J. ACM 30 (1983) 323–342.
[8] S.D. Brookes, A semantics for concurrent separation logic, in: P. Gardner, N. Yoshida (Eds.), CONCUR 2004 – Concurrency Theory, 2004, pp. 16–34.
[9] C. Calcagno, P.W. O'Hearn, H. Yang, Local action and abstract separation logic, in: LICS, IEEE Computer Society, 2007, pp. 366–378.
[10] P.M. Deniélou, N. Yoshida, Dynamic multirole session types, in: T. Ball, M. Sagiv (Eds.), POPL, ACM, 2011, pp. 435–446.
[11] P.M. Deniélou, N. Yoshida, Multiparty session types meet communicating automata, in: H. Seidl (Ed.), ESOP, Springer, 2012, pp. 194–213.
[12] R. Dockins, A. Hobor, A.W. Appel, A fresh look at separation algebras and share accounting, in: Z. Hu (Ed.), APLAS, Springer, 2009, pp. 161–177.
[13] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J.R. Larus, S. Levi, Language support for fast and reliable message-based communication in Singularity OS, in: EuroSys, ACM, 2006, pp. 177–190.
[14] A. Francalanza, J. Rathke, V. Sassone, Permission-based separation logic for message-passing concurrency, CoRR, arXiv:1106.5128, 2011.
[15] M. Giunti, A type checking algorithm for qualified session types, in: L. Kovács, R. Pugliese, F. Tiezzi (Eds.), WWV, 2011, pp. 96–114.
[16] M. Giunti, V.T. Vasconcelos, A linear account of session types in the pi calculus, in: P. Gastin, F. Laroussinie (Eds.), CONCUR, 2010, pp. 432–446.
[17] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, M. Sagiv, Local reasoning for storable locks and threads, in: APLAS, 2007, pp. 19–37.
[18] M. Gouda, E. Manning, Y. Yu, On the progress of communication between two finite state machines, Inf. Control 63 (1984) 200–216.
[19] A. Hobor, C. Gherghina, Barriers in concurrent separation logic, in: G. Barthe (Ed.), ESOP, Springer, 2011, pp. 276–296.
[20] K. Honda, V.T. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programming, in: ESOP, 1998, pp. 122–138.
[21] R. Hu, N. Yoshida, K. Honda, Session-based distributed programming in Java, in: J. Vitek (Ed.), ECOOP, Springer, 2008, pp. 516–541.
[22] G.C. Hunt, J.R. Larus, Singularity: rethinking the software stack, Oper. Syst. Rev. 41 (2007) 37–49.
[23] N. Kobayashi, A partially deadlock-free typed process calculus, ACM Trans. Program. Lang. Syst. 20 (1998) 436–482.

[24] K.R.M. Leino, P. Müller, J. Smans, Deadlock-free channels and locks, in: Proceedings of ESOP 2010, Paphos, Cyprus, March 20–28, 2010, pp. 407–426.
[25] É. Lozes, J. Villard, Reliable contracts for unreliable half-duplex communications, in: M. Carbone, J.M. Petit (Eds.), WS-FM, Springer, 2011, pp. 2–16.
[26] É. Lozes, J. Villard, Shared contract-obedient endpoints, in: M. Carbone, I. Lanese, A. Silva, A. Sokolova (Eds.), ICE, 2012, pp. 17–31.
[27] M. Merro, Locality in the pi-calculus and applications to distributed objects, Ph.D. thesis, Ecole des Mines de Paris, 2000.
[28] P.W. O'Hearn, Resources, concurrency and local reasoning, in: P. Gardner, N. Yoshida (Eds.), CONCUR 2004 – Concurrency Theory, 2004, pp. 49–67.
[29] M. Raza, P. Gardner, Footprints in local reasoning, Log. Methods Comput. Sci. 5 (2009).
[30] K. Takeuchi, K. Honda, M. Kubo, An interaction-based language and its typing system, in: C. Halatsis, D.G. Maritsas, G. Philokyprou, S. Theodoridis (Eds.), PARLE, Springer, 1994, pp. 398–413.
[31] A.J. Turon, M. Wand, A resource analysis of the pi-calculus, CoRR, arXiv:1105.0966, 2011.
[32] J. Villard, Heaps and hops, Ph.D. thesis, École Normale Supérieure de Cachan, 2011.
[33] J. Villard, É. Lozes, C. Calcagno, Proving copyless message passing, in: APLAS, 2009, pp. 194–209.