Shared Contract-Obedient Endpoints

Étienne Lozes
Universität Kassel, Germany

Jules Villard
University College London, UK

Most of the existing verification techniques for message-passing programs suppose either that channel endpoints are used in a linear fashion, where at most one thread may send or receive from an endpoint at any given time, or that endpoints may be used arbitrarily by any number of threads. The former approach usually forbids the sharing of channels while the latter limits what is provable about programs. In this paper we propose a midpoint between these techniques by extending a proof system based on separation logic to allow sharing of endpoints. We identify two independent mechanisms for supporting sharing: an extension of fractional shares to endpoints, and a new technique based on what we call reflexive ownership transfer. We demonstrate on a number of examples that a linear treatment of sharing is possible.

Introduction

One way to tackle the formal verification problem for message-passing programs is to prove that they implement protocols expressible in a higher-level formalism and easier to reason about. Two main approaches coexist: session types on the one hand [21], used to police interactions in programs expressed either in the π-calculus [15] or in a message-passing variant of Java [13], and *channel contracts* on the other hand [6], used for instance to describe the protocols in the Sing# programming language [7] developed for the Singularity operating system [14]. High-level protocol descriptions allow the program verification effort to be split between checking properties of the protocol itself, such as the absence of message reception errors (occurring when a message of the wrong type is received) or of orphan messages (those sent at some point in the interaction but never received before the channel is closed), and checking obedience of the program to the protocol.

Most of the existing verification techniques for message-passing programs suppose 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. Indeed, checking the conformance to a given protocol locally for each thread or process often gives unsound results if this linearity condition is broken. This limitation prevents some programs from being proven. For instance, such linear channel communications are known to enforce a form of determinism [8] that excludes standard synchronisation primitives such as locks and semaphores. Almost none of the verification techniques supporting shared channels is able to give meaningful protocols to them.

This work extends a previous approach based on a marriage of separation logic and channel contracts [25], which forbade any active sharing of endpoints, to deal with endpoint sharing and give meaningful protocols to their interactions. As in previous work, we consider a simple imperative language that features explicit memory manipulation and primitives for creating and destroying bi-directional, asynchronous channels made of two endpoints, and for sending and receiving messages on individual endpoints. Each message is composed of a user-defined label (describing the kind of payload supposed to be transferred) and zero or more values. Crucially, values may be pointers to memory cells or even to other endpoints, and thus sending a message may create sharing of resources and foster data races.

M. Carbone, I. Lanese, A. Silva, A. Sokolova (Eds.):
5th Interaction and Concurrency Experience (ICE'12).
EPTCS 104, 2012, pp. 17–31, doi:10.4204/EPTCS.104.3

© É. Lozes & J. Villard
This work is licensed under the
Creative Commons Attribution License.

In the absence of sharing, our previous work was able to prove obedience to channel contracts and absence of data races, and from that to deduce the absence of reception errors and of orphan messages. In this setting, transferring a message that is attached to a resource is reflected in the logic by transferring ownership of that resource (the message's *footprint* in memory) to the recipient of the message.

We propose an extension to the previous proof system that deals with endpoint sharing and 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*; on the other hand, ensuring the absence of data races. Our notion of data race freedom allows shared read access to memory cells, and, as often as possible, shared access to endpoints for sending and receiving. The latter significantly complicates the analysis of communications, and in particular one of the primary goals of the proof system: to check whether channels obey their declared contracts.

In order to guarantee that shared endpoints obey their contracts, we extend the previous proof system [25] in two ways. The first extension is based on fractional shares [5, 4], a widespread refinement of the all-or-nothing ownership in separation logic. Permissions allow communications on endpoints that are partially owned, with some restrictions as we shall see. Sharing contract-obedient endpoints with permissions is indeed problematic as a consistent view of the contract state of the endpoint should be maintained amongst all sharers. A correct permission-based sharing thus turns out to be extremely close in spirit to unrestricted types by Giunti and Vasconcelos [10] if slightly more flexible (see the related works section). Permission-based sharing still excludes a lot of interesting protocols. Consider for instance the following code featuring a seller and several buyers running in parallel. The seller bargains the price of its product using endpoint e while several buyers concurrently access the peer f of e:

```
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);
}
```

Here `product_descr` and `offer` are *labels* identifying the kind of message being transferred. A natural contract for the communication channel (e, f) is $C = !\texttt{product_descr}.?\texttt{offer}.C$, where ! is used for sending and ? for receiving from the point of view of the first endpoint, the other endpoint following the dual contract where ! and ? are swapped (notice that contracts are only concerned with the labels and directions of the messages being exchanged and not their values). However, the above program is not provable using fractional permissions or unrestricted types and the reason, as we shall see, is that this contract contains two distinct states.

The second extension of our proof system precisely addresses this problem. The crucial step towards proving this seller/buyers example is to consider that none of the buyers makes any assumption on the contract state of f at the beginning of the protocol. In particular, they attempt to receive a product description on f without relying on f being in a contract state that ensures that such a message will be available next. The crucial argument that validates this reasoning is that the received message justifies *a posteriori* the buyer waiting for a `product_descr` message. In other words, the knowledge (or ownership) of the contract state of f is established in the post state of receiving f but not in the pre state: the footprint of this message contains the endpoint that it is received on.

We fully formalise the two extensions of our proof system in the companion technical report [18]. In particular, we give a formal proof of soundness that ensures several safety properties for proved programs. Due to space limitations we have chosen to keep the presentation rather informal, focusing on

example programs, and we invite the interested reader to refer to our technical report for more details on the formalisation.

Outline In the first section we recall our previous model of Sing# and define some of the main problems of interest for verification. In the second section we give a tutorial introduction to the kind of annotations that are used in our proof system. In the third section we illustrate the principle of permission-based sharing of endpoints on a few examples. In the fourth section we introduce the principle of linear sharing with several examples of non-confluent concurrent programs. In the fifth section we give some highlights of the formalisation of our proof system. We conclude with related works.

1 Copyless Message Passing

1.1 Toy Programming Language

We consider a programming language which offers some basic support for procedures, variables, and threads. Threads share a global memory and may allocate, deallocate, and manipulate heap objects allocated in this shared memory. Moreover, threads may synchronise thanks to message exchanges across heap-allocated communication channels. Communications are asynchronous and channels can be thought of as pairs of perfect FIFO buffers. More precisely, channels always consist of exactly two endpoints and each endpoint can be used to send to and receive from the other one. A send instruction `send(m,e,`$v_1,\ldots,v_n$`)` triggers the emission of a message tagged with a label `m` emitted on an endpoint `e` with n values v_1, ..., v_n (the arity of the message is fixed by the tag `m`). A receive instruction `(`$\mathtt{x}_1,\ldots,\mathtt{x}_n$`) = receive(m,e)` is symmetric and specifies which message tag is expected. This communication model is close to the one of Sing# [7].

Example 1.1 *The two threads below collaboratively close the channel they use to communicate. The variables* `e` *and* `f` *are global.*

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

```
put() {
 send(endpoint,e,e);
}
```

```
get() {
 local x;
 x = receive(endpoint,f);
 close(x,f);
}
```

The main function allocates the two endpoints `e`, `f` *of the channel with* `open` *and gives one to each of the two threads* `put` *and* `get` *that it launches in parallel. It then joins both threads. The* `put` *thread sends a message tagged as "*`endpoint`*" to the* `get` *thread. Observe that* `e` *is both the subject and the object of this message: endpoints are heap-allocated and the address of an endpoint is a valid value for a message. Upon reception the* `get` *thread holds both endpoints of the channel and can thus safely close the channel with* `close`.

Our toy programming language also features primitives for heap manipulation. For simplicity, we will assume primitive heap cells with only two fields. We use the syntax `x = new()`, `dispose(x)`, `x.`i` = ...`, and `y = x.`i for allocation, deallocation, destructive update, and look-up ($i \in \{0,1\}$ denotes the field selector). Although endpoints are also heap cells, they are of a different type and we only consider programs that are well-typed in that respect (the proof system will actually verify that this is the case). Message values can contain the memory address of an endpoint or a cell and thus both kind of heap

objects can be passed by message exchanges. The grammar below summarises some of the key aspects of the syntax of our programming language.

```
c ::= skip | x = e |
      x = new() | x.0 = y | x.1 = y | x = y.0 | x = y.1 | dispose(x) |
      (x,y) = open(C) | close(x,y) | send(m,x,x1,...,xn) | (x1,...,xn) = receive(m,x) |
      switch { ... case xi = receive(mi,ei): pi ... } | ...

p ::= c | p; p | p||p | while (b) p | if (b) then p else p | local x; p | ...
```

Example 1.2 *In the code below the* `put` *thread non-deterministically chooses either to send a cell or to keep it. On the other side, the receiver (*`get`*) needs to be ready to receive any of the two messages* `cell` *and* `no_cell`*; this is achieved by the* `switch`/`case` *construct. Variables not declared as local are global.*

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

```
put() {
 local f;
 if (...) {
  send(cell,e,x);
 } else {
  send(nocell,e);
  dispose(x);
 }
 f = receive(clos,e);
 close(e,f);
}
```

```
get() {
 switch {
 case y = receive(cell,f):
  dispose(y);
 case receive(nocell,f):
  skip;
 }
 send(clos,f,f);
}
```

1.2 Pitfalls of Copyless Message Passing

Programs may suffer from several runtime errors. First of all, *memory violations* may occur when a dangling pointer is dereferenced, modified, or disposed. Moreover, two threads may cause a *data race* if they simultaneously try to access a variable or a memory location and if at least one of the accesses is a write access. Synchronisations via message passing can be used to prevent such races but might yield *deadlocks*: in this paper, a program is considered to be in a deadlock configuration if all of its threads are blocked on receptions.

In addition to these rather familiar problems, it is important to mention other erroneous behaviours one may wish to avoid. The first one is *memory leaks* (situations where it becomes impossible to fully deallocate the heap), which one may want to detect or forbid altogether, either because the underlying programming language is not garbage collected or because this information may matter for the garbage collector (the latter holds for Sing#). Similarly, sent messages may never be received and stay in a channel forever; closing channels when they contain such *orphan messages* is most of the time considered a communication error. Instead, one should make sure that both queues of a channel are empty before closing it. Lastly, a thread may enter a `switch`/`case` construct in a situation where the buffer of one of the endpoints it scans starts with a message whose tag is not listed as a possible case. We strive to ensure the absence of such *unspecified receptions*, though they might be resolved differently in other communication models (for instance the program may block or look for a message with one of the listed tag deeper in the queue, *à la* Erlang).

Example 1.3 *The program* $P_0 \triangleq$ `send(m1,e) || switch {case receive(m2,f): skip}` *is both deadlocking and triggering an unspecified reception. Similarly, replacing* `get()` *by* `skip` *in Example 1.1 would cause a memory leak.*

Note that all of these problems are distinct and incomparable. In particular,

- orphan messages are not a special case of memory leaks: it may be the case that lost messages carried no ownership (*e.g.* the message `nocell` above);
- unspecified receptions are not a special case of deadlocks: we will distinguish the program $P_1 \triangleq$ `send(m1,e) || receive(m2,f)` from the program P_0 above: P_1 is only deadlocking.[1] Conversely, the program `P0 || {receive(m1,f); send(m2,e);}` is not deadlocking. However, it has an unspecified reception at the time `m1` is sent.

2 Proving Copyless Message Passing

2.1 Separation Logic

Our aim is to design a proof system for Hoare triples based on separation logic that will prevent all of the aforementioned errors except deadlocks[2], as was achieved previously in a context without endpoint sharing [25]. Hoare triples are understood in terms of partial correctness and, for this reason, non-terminating and deadlocking programs generally have a proof. A triple $\{A\}\ p\ \{B\}$ is informally understood as "starting p from any state satisfying A will not result in a memory fault, an unspecified reception or an orphan message and, if p terminates, then the final state satisfies B".

Being based on separation logic, our proof system will ensure that every proved program follows some locality principles. As a first approximation (and before we introduce permissions), these locality principles can be summarised as follows.[3]

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

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

These principles can be illustrated on the program of Example 1.1: observe that at every point in time, `put` and `get` own disjoint parts of the heap. Initially, `put` owns the endpoint `e` while `get` owns the endpoint `f`. During execution, the ownership of `e` is transferred from `put` to the message `endpoint` and ultimately to `get`, which justifies its disposal by `get`.

State assertions make this reasoning more formal. They convey information on the part of the heap that is owned by the thread at a given point in time. The base predicates of state assertions describe the ownership of a single cell or endpoint.

The example code on the left-hand side of Figure 1 features the annotations that formalise the explanations given above. Separation logic formulas appear in square brackets. In the precondition of `main`, emp indicates that nothing is known to be allocated (or, alternatively, that nothing in the heap is owned at this program point). After the `(e,f) = open()` instruction, two peer endpoints have been allocated and their addresses have been stored in variables `e` and `f` respectively. The predicate $e \mapsto\!\!\!\Rightarrow f$ denotes the

[1] Provided no other thread accesses `f` concurrently. Note that one may wish to authorise a thread to wait for a message in a buffer provided that other threads will pick all the other ones first. In order to support such behaviours we distinguish reception from scanning (also called peeking) and consider that unspecified receptions might occur only on scanning, not on reception.

[2] The reason why we omit deadlocks is not that we find them uninteresting, but rather that there is no simple way to treat them without introducing new notions; see Leino et al. [16] or session types [15] for analyses of message-passing programs dealing with deadlocks.

[3] The ownership hypothesis and separation property also apply to global variables. Since this is cumbersome to formalise we leave that aspect implicit in this paper.

```
main() [emp] {
  (e,f) = open();
  [e⇰f * f⇰e]
  [e⇰f]   ||  [f⇰e]
  put()   ||  get();
  [emp]   ||  [emp]
} [emp]
```

```
multi_readers() [emp] {
  x = new();
  [x ↦ (−,−)]
  x.1 = 7;
  [x ↦ (−,7)]
  [x ↦0.5 (−,7)]  ||  [x ↦0.5 (−,7)]
  z = 1 + x.1     ||  t = 3 × x.1
  [x ↦0.5 (−,7)]  ||  [x ↦0.5 (−,7)]
  [x ↦ (−,7)]
  dispose(x);
} [emp]
```

Figure 1: Separation logic at work: state splitting and parallel composition.

```
message endpoint(x) [x⇰f]

put() [e⇰f] {
  send(endpoint,e,e);
} [emp]
```

```
get() [f⇰e] {
  local x;
  x = receive(endpoint,f);
  [f⇰e * x⇰f]
  [f⇰x * x⇰f]
  close(x,f);
} [emp]
```

Figure 2: Proof sketches for `put` and `get`.

ownership of the endpoint e and provides the information that its peer is f. In the second assertion, the separating conjunction $*$ is used to add together the two disjoint pieces of owned heap consisting each of one endpoint. The proof of the parallel composition distributes the pieces of heap currently owned to each of the sub-threads and merges them back at the end of the parallel composition.

The main limitation of this treatment of parallelism is that it excludes programs based on the multiple-readers/single writer principle. One solution is to assign to every owned piece of heap a permission $\pi \in (0,1]$. The permission 1 is the "total" or "write" permission while a permission $\pi < 1$ is a "partial" or "read" permission. To illustrate this idea, consider the example code on the right-hand side of Figure 1. Initially, the program allocates a new cell x for which it may assume a write permission. It results in a state where the ownership of the cell x with permission 1 can be assumed, which is denoted by $x \mapsto (-,-)$. It is then allowed to update the content of the cell, resulting in $x \mapsto (-,7)$.[4] At the level of the parallel composition, we split the write permission to distribute it on each side, resulting in two read permissions which we choose to be 0.5 each (uneven splittings would also have been possible), for which we write $x \mapsto_{0.5} (-,7)$.[5] On each side, cell updates are now forbidden (note that all sharers may in return assume that the value of the second field does not change) but shared reads are allowed. After the parallel step, all permissions are collected back and a write permission for the cell is again granted which allows us to dispose the cell.

To conclude this overview, let us complete the proof of the `put` and `get` functions of Example 1.1. The *footprint* of a message is the piece of heap that is lost when sending this message and gained when receiving it.[6] The correctness of the interaction between `put` and `get` is based on the assumption that the footprint of the `endpoint` message is the endpoint `e`, represented by the formula $e \Mapsto f$. More generally,

[4] We write $x \mapsto (v_1,v_2)$ for a cell x whose first and second fields contain the values v_1 and v_2. We freely use a wildcard "–" when a value is existentially quantified. Note that we use different arrows for regular memory cells ($\mapsto$) and endpoints ($\Mapsto$).

[5] We explicitly specify the subscript permission only when it is different from 1.

[6] We use a different terminology than previous work [25] where footprints were called *message invariants*.

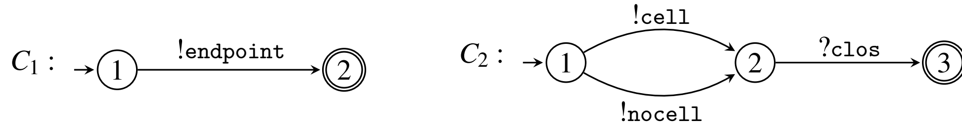
given a program to prove, we will associate a footprint description to every user-defined message tag m and assume that every time m is sent or received the same footprint is transferred. Figure 2 shows the state assertions at each program point for `put` and `get`, which constitute a proof sketch. Note that before closing the channel the proof makes the deduction step that $f \mapsto e * x \mapsto f \vdash f \mapsto x * x \mapsto f$, which is valid because each endpoint is peered to exactly one other endpoint.

2.2 Channel Contracts

Separation logic naturally enforces some of the safety conditions we listed in Section 1.2, such as data race freedom. However, in order to detect either an orphan message or an unspecified reception, proofs must support assumptions about the behaviour of the environment of a thread. In particular, one should be able to specify which messages local endpoints *must* be ready to receive and which messages they *may* send.

Channel contracts are a specialised form of rely/guarantee reasoning that precisely addresses this problem; they describe the protocol followed by each endpoint as a finite-state automaton. A contract C is written from one of the endpoints' point of view, the other endpoint being assumed to follow the dual contract $\bar{C}$ where sends ! and receives ? have been swapped.

Example 2.1 *The contracts for the two previous example programs are depicted below.*



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

Contracts are used to "type" endpoints: at any point in time, every endpoint is decorated with a contract and a contract state. The contract is fixed for the whole life of the endpoint, from `open` to `close`. The contract state evolves along the communications: if `e` is in the contract state q of its contract C and if a message m is sent over `e`, then the contract state of `e` becomes the $!m$ successor of q in C. If q does not have such a successor, then the contract is violated. Similarly, in case of a reception of m the contract state evolves to its $?m$ successor. A `switch`/`case` construct on a given endpoint should be ready to handle at least all the receptions indicated by the contract in the current control state of the endpoint. Peer endpoints `e`, `f` allocated with `open(`C`)` are ruled by contracts C and $\bar{C}$ respectively and start in the initial contract state of C. Finally, `close(e,f)` is only allowed when both endpoints are peers of each other and are in the *same* final state of their contract (contracts may have multiple final states that way). Note that contracts without final states describe persistent channels that cannot be closed.

Example 2.2 *The proof that the endpoint-transferring program of Example 1.1 obeys the contract C_1 of the Example 2.1 is sketched below. Annotations now feature the predicate $e \mapsto (C\langle q \rangle, f)$ that denotes the ownership of an endpoint e ruled by the contract C, currently in state q of C, and whose peer is f.*

```
message endpoint(x) [x ↦ (C1⟨2⟩,f)]
put() [e ↦ (C1⟨1⟩,f)] {
 send(endpoint,e,e);
} [emp]
```

```
main() [emp] {
 (e,f) = open(C);
 put() || get();
} [emp]
```

```
get() [f ↦ (C̄1⟨1⟩,e)] {
 local x;
 x = receive(endpoint,f);
 close(x,f);
} [emp]
```

In general, contracts do not guarantee the absence of communication errors or of orphan messages. Moreover, they are known to be Turing powerful so deciding most properties of interest is an undecidable problem [17]. However, sufficient conditions exist that ensure their correctness: if contracts are deterministic automata with no mixed choice (all outgoing transitions from a given state are either all sends or all receives), then receptions errors are prevented. If moreover every cyclic path containing a final state contains at least one send transition and one receive transition, then there cannot be orphan messages upon channel closure [17]. All the contracts in this paper satisfy these restrictions.

3 Contract Obedience beyond Linearity

Permissions support scenarios involving sharing among multiple readers. Until now we only saw how permissions apply to standard heap cells, but not endpoints. Let us introduce a predicate $e \mapsto_\pi (C\langle q\rangle, f)$ that denotes the ownership of a fraction π of the endpoint e. This extension immediately raises a question: what permission do we require to send, receive, and close? A conservative solution which would again prevent any active sharing would be to require a full permission in all cases. A rather naive analogy with heap cells could suggest to ask for a full (write) permission for `send` and `close` but only a partial (read) permission for receptions. Were we concerned with routing messages deterministically we could conversely allow partial permission for sending but require full permissions for receiving. Our solution is more permissive than both of these: we only require full permissions to close channels.

This choice comes with some restrictions, lest the proof system becomes unsound. When two threads concurrently send (or receive) the messages m_1 and m_2 respectively over a shared endpoint `e` they may change the contract state of `e` in different ways and get inconsistent views of the actual endpoint's state. Would it be safe if we were to restrict ourselves to situations where $m_1 = m_2$? The answer is no and is worth being illustrated by an invalid proof. Consider a cell-transferring protocol with two producers sharing an endpoint `e` ruled by the contract $C = \rightarrow (1) \xrightarrow{!\texttt{cell}} ((2))$.

```
message cell(x) [x ↦ −]
put() [e⇛0.5 (C⟨1⟩,f)] {
  local x; x = new();
  send(cell,e,x);
} [e⇛0.5 (C⟨2⟩,f)]
```

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

```
get() [f⇛(C̄⟨1⟩,e)] {
  local y;
  y = receive(cell,f);
  dispose(y);
} [f⇛(C̄⟨2⟩,e)]
```

This program should be rejected: there is an orphan message and a memory leak when the channel is closed. However, the invalid proof above incorrectly establishes the Hoare triple $\{\mathsf{emp}\}$ `main()` $\{\mathsf{emp}\}$. Moreover, both concurrent `put` functions update the contract state of `e` to the same state 2.

Let us think about the problem differently and consider that endpoints are typed at run-time (which is the case for Sing#), or in other words that the contract state is a field of the heap cell implementing the endpoint. Following the same principle as for standard heap cells, we should only allow updates to the contract state of an endpoint when it is owned with the full write permission. In the following we only consider proofs that follow this principle. Excluding updates of the contract state of a shared endpoint may seem to forbid any communication on this endpoint. There is however a very particular case where a `send` or `receive` does not generate a contract state update: a self-loop on a contract state.

Example 3.1 *Consider the contract C depicted in Figure 3. Then a shared endpoint in state* 1 *can be used to send a cell message. The two producers / one consumer program of Figure 3 can thus be proved.*

Note how sending and receiving `clos` only occurs with the write permission on the endpoint, whereas `send(cell,e,x)` occurs with just a read permission over `e`. Crucially, the `cell` message silently carries

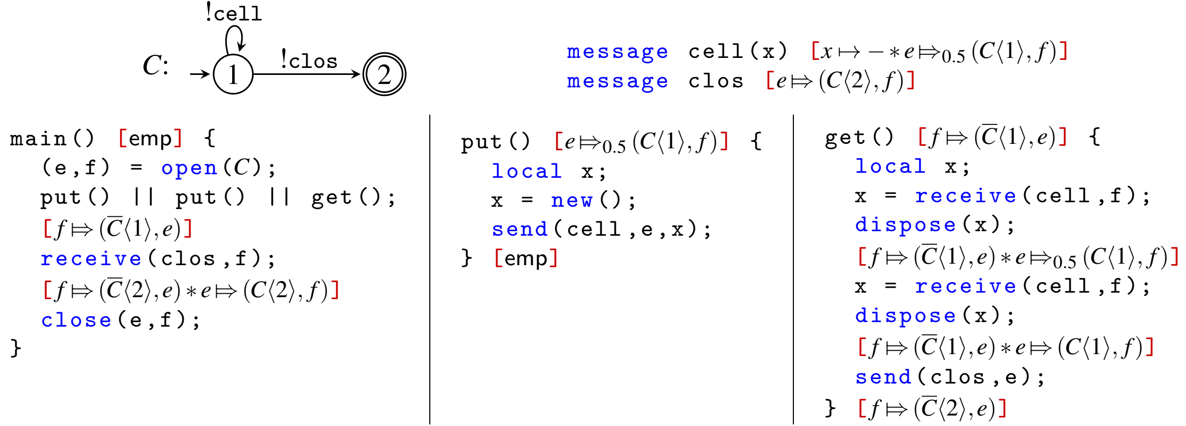


Figure 3: Two producers and one consumer: non-linear sharing.

half of the ownership of the endpoint `e`. After receiving a `cell` message twice the consumer has full ownership of `e` and can thus send the `clos` message in a contract-obedient way. Note that we cannot avoid the transfer of `e`: were the `clos` message sent from `f`, the contract state 1 would be a mixed state.

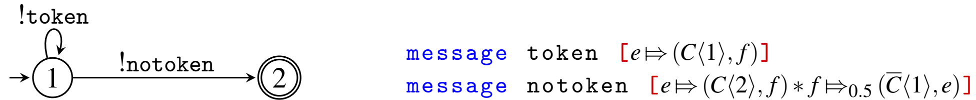
Example 3.2 (Internal Choice) *A slightly more involved example is the following encoding of internal choice. We let two processes compete to get a* `token` *message. The one that catches it first can start executing and sends a* `notoken` *message to the other process, which is in charge of closing the channel that carried the token.*

$$choice(p_1, p_2) \triangleq \texttt{(e,f) = open}(C)\texttt{; \{ send(token,e) || } p_1 \texttt{ || } p_2 \texttt{ \}}$$

where `e`, `f` *do not appear free in p_1, p_2 and p'_i is defined parametrically in the program p_i by*

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

Let us derive the Hoare triple $\{A\}$ $choice(p_1, p_2)$ $\{B\}$ from $\{A\}$ p_1 $\{B\}$ and $\{A\}$ p_2 $\{B\}$ for any formulas A and B. We first give to each process half of the ownership on the endpoint `f`*. Then, we consider that the ownership of the endpoint* `e` *is transferred by the messages* `token` *and* `notoken`*. Moreover, we set the message* `notoken` *to transfer half of the ownership of* `f`*, so that the loser of the internal choice gets the full ownership on the two endpoints in the end and can close the channel.*



4 A Linear Form of Sharing

Sharing endpoints is very common in scenarios involving one server and several clients: the server usually owns an endpoint, say `ep`, and the clients share its peer, say `ip`. One way to model a client connection is to assume that the server first sends a `welcome` message. Once caught by one of the clients, a new pair of endpoints is used for the rest of this connection. The code in Figure 4 models such negotiated connections among a central server and several clients, letting the clients allocate the endpoints used for

Clients and Server

```
server(){
  local e;
  ep = bind(ip);
  listen();
  while (...) {
    e = accept();
    spawn serve(e);
  }
  close(ip,ep);
}

client(){
  local f;
  f = connect();
  start_service(f);
}
```

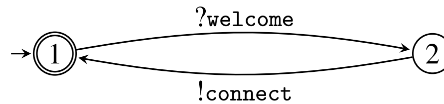
"System calls"

```
listen(){ send(welcome,ep); }

accept(){
  local e;
  e = receive(connect,ep);
  send(welcome,ep);
  return e;
}

connect(){
  local e,f;
  receive(welcome,ip);
  (e,f) = open(Service);
  send(connect,ip,e);
  return f;
}
```

Figure 4: Negotiating connections between a server and several clients.

their connection.[7] For added realism, the code assumes a primitive `bind` that takes a "spinster" endpoint *ip* and peers it to a new endpoint. It could be replaced by an `open` primitive in an initial phase before the clients and the server are spawned. The `spawn` construct may be simulated using $\|$ and recursion.

Observe that the `welcome` message carries no endpoint for the service, but rather grants the right to the client that receives it to initiate a service. The server thus lets the client allocate the two endpoints for the service. One of them is sent back to the server in the `connect` message. Overall, the connection contract *Service* is



Such a contract is problematic for permission-based sharing, as all transitions have a different source and target state. As we saw in the previous section, we should not allow communications on partially owned endpoints in that case.

The problem may be narrowed down to the `connect` function, where the client first receives on *ip* and then sends on it. Clearly, the client needs to own the endpoint *ip* with full permission in the state before the send, because at this time it has to change the endpoint's state. Since other clients may be in the initial state of the `connect` function at the same time, not even a read permission on the endpoint *ip* can be granted in the pre state of the `connect` function. Between the two states the message `welcome` is received; this seems like a good place to acquire the ownership of *ip*. But shouldn't the client own *ip* before it receives the `welcome` message on it? It indeed needs to update the contract state of *ip* as well.

The answer is no; let us explain why. The `receive` instruction needs indeed the ownership of *ip* but not necessarily in the pre state. Reasoning backwards gives the explanation. In a correct proof we would have $\{Pre\}$ `receive(welcome,ip)` $\{Post\}$ for some $Pre, Post$ formulas. Since the reception updates the contract state of *ip*, *Post* should be of the form $ip \mapsto (\overline{C}\langle 2\rangle, ep) * \ldots$ Let F denote the footprint of the `welcome` message and let $Post' = ip \mapsto (C\langle 1\rangle, ep) * \ldots$ denote the weakest precondition of *Post* before

[7] The client often uses only few services of the server. For this reason, it could be expected that allocating endpoints on the client's side achieves better performances, especially if the buffer's size drastically shrinks between the client's specific contract and a general-purpose service contract implemented by the server.

updating the contract state. Then we should have that $Pre * F$ entails $Post'$. As it turns out, nothing more is required and the ownership of $ip \mapsto\!\!\!> (C\langle 1\rangle, ep)$ may just as well be granted by the footprint F of the message. In other words, the ownership of the endpoint ip can be granted *a posteriori* by the reception of the `welcome` message over ip.

Let us present the formal proof of the `connect` function using this principle. We will soon present the formal rules supporting this reasoning and refer to our technical report for a proof of soundness.

```
message welcome [ip ⇰ (C⟨1⟩,ep)]
message connect(e) [ip ⇰ (C⟨2⟩,ep) ∗ e ⇰ −]
```

```
connect() [emp] {
  local e,f; receive(welcome,ip);
  [ip ⇰ (C⟨2⟩,ep)]
  (e,f) = open(Service);
  [ip ⇰ (C⟨2⟩,ep) ∗ e ⇰ f ∗ f ⇰ e]
  send(connect,ip,e); return f;
} [f ⇰ e]
```

This way of reasoning might seem unorthodox since it breaks the (false) intuition that shared channels are not linear. Perhaps surprisingly, the above example is indeed a case of sharing channels in a purely linear fashion. Before we formalise the way that our proof system works, let us illustrate the strength of this reasoning principle on a last example (see also our technical report for examples with multicast and synchronisation barriers).

Example 4.1 (Locks) *We develop now an encoding of heap-allocated locks. Our design choices closely follow the ones of Gotsman et al. [11].*

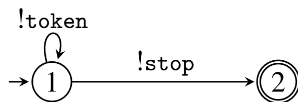
We introduce a macro predicate $\mathsf{handle}_\pi(x)$ *(*$\pi \in (0,1]$*) that denotes the right to attempt acquiring or releasing a lock x. The ownership of the handle is granted with full permission when the lock is allocated and it is required with full permission as well when the lock is disposed, whereas only a read permission is required to either acquire or release the lock. We also introduce a macro predicate* $\mathsf{locked}(x)$ *that holds when the lock is locked by the current thread. It is assumed whenever the lock is acquired and lost when the lock is released. We consider non-reentrant locks, i.e. the formula* $\mathsf{locked}(x) * \mathsf{locked}(x)$ *is unsatisfiable (other design choices are possible).*

The lock intends to protect a piece of heap that satisfies a certain invariant I. When the lock is acquired the ownership of I should be ∗-conjoined and when it is released I should be consumed. Like Gotsman et al. [11], we assume that the lock is initially acquired by the thread that creates it and that it can only be released by a thread that has acquired the lock. We propose the following encoding of locking primitives:

```
new_lock(x) [emp] {
  x = new();
  (x.0,x.1) = open(C);
} [handle(x) ∗ locked(x)]

dispose_lock(x) [handle(x) ∗ locked(x)] {
  send(stop,x.0); receive(stop,x.1);
  close(x.0,x.1); dispose(x);
} [emp]
```

```
acquire(x) [handle_π(x)] {
  receive(token,x.0);
} [handle_π(x) ∗ locked(x) ∗ I]

release(x) [handle_π(x) ∗ locked(x) ∗ I] {
  send(token,x.1);
} [handle_π(x)]
```

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.*




```
message token [locked(x) ∗ I]
message stop [emp]
```

All messages transit through the endpoints $x.0$ and $x.1$, which are thus shared among all client threads. More precisely, the ownership of the cell x is shared by means of read permissions but the ownership of the endpoints is shared linearly using the same form of backward reasoning as in the previous example. The `token` *message thus transfers the write ownership of $x.0$ and $x.1$. and gives the right to receive the* `token` *message in the* `acquire` *function even if the endpoint $x.0$ is not owned in the pre state. The macro predicates are defined as follows:*

$$\begin{array}{lcl} \mathsf{locked}(x) & \triangleq & \exists X_0, X_1 .\ x \mapsto_{0.5} (X_0, X_1) * X_0 \mapsto (C\langle 0\rangle, X_1) * X_1 \mapsto (\overline{C}\langle 0\rangle, X_0) \\ \mathsf{handle}_\pi(x) & \triangleq & x \mapsto_{\pi/2} (-,-) \end{array}$$

5 Proof System

5.1 Proof Rules

Our proof system is defined by a set of inference rules for Hoare triples. The core of the proof system is composed of the standard proof rules of concurrent separation logic [20] which are omitted here due to space restrictions. Let us briefly comment the new rules for copyless message passing.

Channel allocation and deallocation The rules for allocation and disposal are rather symmetric: `open` produces two peered endpoints whereas `close` consumes them.

$$\frac{q_0 = \mathsf{init}(C)}{\{\mathsf{emp}\}\ \texttt{(e,f) = open}(C)\ \{e \mapsto (C\langle q_0\rangle, f) * f \mapsto (\overline{C}\langle q_0\rangle, e)\}}$$

$$\frac{q \in \mathsf{final}(C)}{\{e \mapsto (C\langle q\rangle, f) * f \mapsto (\overline{C}\langle q\rangle, e)\}\ \texttt{close(e,f)}\ \{\mathsf{emp}\}}$$

Notice that `open` allocates two endpoints that are peers and in the initial state while `close` requires two peers that are in a same final state. Also note that a full permission is granted and required in each case.

Send and receive The proof rules of `send` and `receive` require the introduction of a pseudo-instruction $\texttt{skip}_{e\lambda,f}$, where λ denotes either $!m$ or $?m$. It has the operational meaning of `skip` but also updates the contract state.

$$\frac{q \xrightarrow{\lambda} q' \in C \qquad q = q' \vee \pi = 1}{\{e \mapsto_\pi (C\langle q\rangle, f)\}\ \texttt{skip}_{e\lambda,f}\ \{e \mapsto_\pi (C\langle q'\rangle, f)\}}$$

Intuitively, $\texttt{skip}_{e\lambda,f}$ 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.

Let us now consider the rules for `send` and `receive`. Let $I_m(x, y_1, \ldots, y_n)$ be the footprint of a message $m(y_1, \ldots, y_n)$ – the x parameter of the footprint formula denotes the endpoint that sends the message. Then $I_m(e, \vec{y})$ is lost upon sending and acquired upon receiving. Either operation updates the contract state accordingly using $\texttt{skip}_{e\lambda,f}$.

$$\frac{\{A\}\ \texttt{skip}_{e!m,f}\ \{B * I_m(e,\vec{x})\}}{\{A\}\ \texttt{send}(m,e,\vec{x})\ \{B\}} \qquad \frac{\{A * I_m(f,\vec{x})\}\ \texttt{skip}_{e?m,f}\ \{B\} \qquad \vec{x} \text{ not free in } A}{\{A\}\ (\texttt{x}_1,\ldots,\texttt{x}_n)\ \texttt{= receive}(m,e)\ \{B\}}$$

Note that in the `send` case the footprint of the message is transferred *after* the endpoint state has been updated. Note also that the footprint may contain the endpoint itself. Examples of such a reflexive ownership transfer are the `clos` message or the `cell` message in Example 3.1 (the latter with half permission).

Conversely, in the `receive` case the footprint of the message is transferred *before* the update of the endpoint's state. The footprint may again contain the endpoint itself. This possibility permits linear sharing of endpoints. Examples of such a reflexive ownership transfer are the `welcome` message of the client-server example, the `token` message of the lock example, or the `notoken` message of the encoding of internal choice (the latter with half permission).

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

$$\frac{\{A\}\ \texttt{switch \{ cases1 \}}\ \{B\} \qquad \{A\}\ \texttt{switch \{ cases2 \}}\ \{B\}}{\{A\}\ \texttt{switch \{ cases1 cases2 \}}\ \{B\}}$$

$$\frac{\mathsf{choices}(C,q) \subseteq \{m_1,\ldots,m_n\} \qquad \{e \mapsto_\pi (C\langle q\rangle, f) * A\}\ \vec{x}_i\ \texttt{=}\ \texttt{receive}(m_i,e)\texttt{:}\ p_i\ \{B\}\ \text{for all } i}{\{e \mapsto_\pi (C\langle q\rangle, f) * A\}\ \texttt{switch \{}\ \ldots\ \vec{x}_i\ \texttt{=}\ \texttt{receive}(m_i,e)\texttt{:}\ p_i\ \ldots\ \texttt{\}}\ \{B\}}$$

5.2 Soundness

Villard defined an operational semantics for our programming language in his PhD thesis [24] and showed the soundness of a previous version of our proof system. The proof easily adapts to the extended version of the proof system (see our technical report [18]).

Many problems complicate the proof of the fact that "well proved" programs are safe. By *safe* we mean that all the erroneous behaviours listed in Section 1.2 are ruled out, except for deadlocks. To be well proved, a program does not just need a proof derivation in our proof system, it also needs well-formed contracts and well-formed message footprints. Firstly, as mentioned in Section 2.2, contracts should ensure the absence of communication errors and of orphan messages. Secondly, we require that footprints be *precise* formulas (a common restriction in concurrent separation logics [23]). Lastly, all of these requirements do not suffice to detect all memory leaks. Sing# adopts the following restriction which prevents them: an endpoint can be sent only when it is in a sending state. This restriction is a relaxed version of Merro's locality condition for the π-calculus [19]. Many of the examples of the paper do not follow Sing#'s restriction: the encoding of internal choice and all the examples of Section 4 use the possibility of sending an endpoint f over its peer e. Generalisations of the Sing# condition include the well-foundedness condition of Bono and Padovani [2] and Villard's condition [24]. In our technical development we use a condition close to the latter. We refer to the technical report for the details of our soundness result and proofs.

6 Related Works

Giunti and Vasconcelos were the first to consider the problem of sharing contract obedient endpoints and extended session types for that purpose by introducing a distinction between linear and unrestricted channels [10]. 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. To retain soundness, unrestrictedly typed channels are limited to "single state" protocols and thus this type system can handle significantly less scenarios than ours. Giunti recently implemented a type-checker for qualified types [9].

Turon and Wand [22] were the first to propose a proof system for message-passing concurrency that exploited permissions. Their proof system addresses the (untyped) π-calculus and introduces some proof rules for temporal reasoning and refinement.

Leino, Mueller and Smans [16] proposed a proof system for reasoning about locks and asynchronous unidirectional channels. Since they don't have contracts, buffers are assumed to transfer only one kind of message. Their work focus instead on preventing deadlocks. They implemented their proof system in the Chalice tool.

Bell, Appel and Walker [1] proposed a proof system for asynchronous unidirectional channels shared by means of permissions. Their proof system ignores contracts and instead describes the contents of channels explicitly and relies on a rule for interleaving the local histories of two endpoints when they get $*$-conjoined.

Bono, Messa, and Padovani [2] introduced a type system *à la* session type for CoreSing#, a model of Sing# very close in spirit to our previous work without sharing [25]. They accurately pointed out that our previous work prevented orphan messages but not memory leaks, and introduced a well-foundedness condition to fix the problem. More recently, they extended their type system with qualified types [3].

Hobor and Gherghina [12] proposed a proof system for synchronisation barriers based on separation logic that was helped by "barrier diagrams", a contract-like description of the sequences of synchronisations on the barrier. We propose an encoding of barriers with sharing in our technical report [18].

Sassone, Rathke and Francalanza [8] introduced a separation logic for CCS and stated a confluence property for the form of linear CCS addressed by their logic. Our proof system shows that, with little effort, it is possible to go past confluent programs while still obeying this kind of discipline.

Merro showed that the full π-calculus can be encoded in the local π-calculus [19]. 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.

References

[1] Christian J. Bell, Andrew W. Appel & David Walker (2010): *Concurrent Separation Logic for Pipelined Parallelization*. *SAS*, *LNCS* 6337, pp. 151–166. Available at `http://dx.doi.org/10.1007/978-3-642-15769-1_10`.

[2] Viviana Bono, Chiara Messa & Luca Padovani (2011): *Typing Copyless Message Passing*. *ESOP*, *LNCS* 6602, Springer, pp. 57–76. Available at `http://dx.doi.org/10.1007/978-3-642-19718-5_4`.

[3] Viviana Bono & Luca Padovani (2012): *Typing Copyless Message Passing*. Logical Methods in Computer Science 8(1). Available at `http://dx.doi.org/10.2168/LMCS-8(1:17)2012`.

[4] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn & Matthew J. Parkinson (2005): *Permission accounting in separation logic*. *POPL*, pp. 259–270. Available at `http://doi.acm.org/10.1145/1040305.1040327`.

[5] John Boyland (2003): *Checking Interference with Fractional Permissions*. *SAS*, *LNCS* 2694, pp. 55–72. Available at `http://link.springer.de/link/service/series/0558/bibs/2694/26940055.htm`.

[6] Daniel Brand & Pitro Zafiropulo (1983): *On Communicating Finite-State Machines*. *J. ACM* 30(2), pp. 323–342. Available at `db/journals/jacm/BrandZ83.html,http://doi.acm.org/10.1145/322374.322380`.

[7] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus & Steven Levi (2006): *Language support for fast and reliable message-based communication in Singularity OS*. EuroSys, ACM, pp. 177–190. Available at `http://doi.acm.org/10.1145/1217935.1217953`.

[8] Adrian Francalanza, Julian Rathke & Vladimiro Sassone (2011): *Permission-Based Separation Logic for Message-Passing Concurrency*. *CoRR* abs/1106.5128. Available at `http://arxiv.org/abs/1106.5128`.

[9] Marco Giunti (2011): *A type checking algorithm for qualified session types*. *WWV*, *EPTCS* 61, pp. 96–114. Available at `http://dx.doi.org/10.4204/EPTCS.61.7`.

[10] Marco Giunti & Vasco Thudichum Vasconcelos (2010): *A Linear Account of Session Types in the Pi Calculus*. *CONCUR*, *LNCS* 6269, pp. 432–446. Available at `http://dx.doi.org/10.1007/978-3-642-15375-4_30`.

[11] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky & Mooly Sagiv (2007): *Local Reasoning for Storable Locks and Threads*. *APLAS*, *LNCS* 4807, pp. 19–37. Available at `http://dx.doi.org/10.1007/978-3-540-76637-7_3`.

[12] Aquinas Hobor & Cristian Gherghina (2011): *Barriers in Concurrent Separation Logic*. *ESOP*, *LNCS* 6602, Springer, pp. 276–296. Available at `http://dx.doi.org/10.1007/978-3-642-19718-5_15`.

[13] Raymond Hu, Nobuko Yoshida & Kohei Honda (2008): *Session-Based Distributed Programming in Java*. *ECOOP*, 5142, Springer, pp. 516–541. Available at `http://dx.doi.org/10.1007/978-3-540-70592-5_22`.

[14] Galen C. Hunt & James R. Larus (2007): *Singularity: rethinking the software stack*. Operating Systems Review 41(2), pp. 37–49. Available at `http://doi.acm.org/10.1145/1243418.1243424`.

[15] Honda Kohei, Thudichum Vasconcelos Vasco & Kubo Makoto (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. *ESOP*, *LNCS* 1381, pp. 122–138. Available at `http://dx.doi.org/10.1007/BFb0053567`.

[16] K. Rustan M. Leino, Peter Müller & Jan Smans (2010): *Deadlock-Free Channels and Locks*. *ESOP*, *LNCS* 6012, pp. 407–426. Available at `http://dx.doi.org/10.1007/978-3-642-11957-6_22`.

[17] Étienne Lozes & Jules Villard (2011): *Reliable Contracts for Unreliable Half-Duplex Communications*. *WS-FM*, *LNCS* 7176, Springer, pp. 2–16. Available at `http://dx.doi.org/10.1007/978-3-642-29834-9_2`.

[18] Étienne Lozes & Jules Villard (2011): *Sharing Contract-Obedient Endpoints*. Research Report LSV-11-23, Laboratoire Spécification et Vérification, ENS Cachan, France. Available at `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/rr-lsv-2011-23.pdf`. 42 pages.

[19] Massimo Merro (2000): *Locality in the pi-calculus and applications to distributed objects*. Ph.D. thesis, Ecole des Mines de Paris.

[20] Peter W. O'Hearn (2004): *Resources, Concurrency and Local Reasoning*. *CONCUR*, *LNCS* 3170, Springer, pp. 49–67. Available at `http://dx.doi.org/10.1007/978-3-540-28644-8_4`.

[21] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-Based Language and its Typing System*. *PARLE*, 817, Springer, pp. 398–413. Available at `http://dx.doi.org/10.1007/3-540-58184-7_118`.

[22] Aaron Joseph Turon & Mitchell Wand (2011): *A resource analysis of the pi-calculus*. *CoRR* abs/1105.0966. Available at `http://arxiv.org/abs/1105.0966`.

[23] Viktor Vafeiadis (2011): *Concurrent Separation Logic and Operational Semantics*. Electr. Notes Theor. Comput. Sci. 276, pp. 335–351. Available at `http://dx.doi.org/10.1016/j.entcs.2011.09.029`.

[24] Jules Villard (2011): *Heaps and Hops*. Ph.D. thesis, École Normale Supérieure de Cachan.

[25] Jules Villard, Étienne Lozes & Cristiano Calcagno (2009): *Proving Copyless Message Passing*. *APLAS*, *LNCS* 5904, pp. 194–209. Available at `http://dx.doi.org/10.1007/978-3-642-10672-9_15`.