Master en Informatique Fondamentale
École Normale Supérieure de Lyon

M2 Internship
Ducruet Léon

Exceptional Actris: Session-Type Based Error Handling in Separation Logic

Abstract :
In this internship, I improved my knowledge about the higher order separation logic framework Iris, and I learned more about the session-type based logic Actris. I implemented a simple message passing library and extended Actris with abortion protocols in order to catch the behaviour of a dropped communication. The results and specifications are implemented in Iris in Coq, and made compatible with the diaframe proof automation tool.

Key words : *Message Passing, Separation Logic, Session Types, Error Handling, Formal Verification.*

Supervised by :
Krebbers Robbers
robbertkrebbers.mail@robbertkrebbers.nl
Radboud University Nijmegen
Houtlaan 4
6525 XZ Nijmegen, Netherlands
https://www.ru.nl/en

June 28, 2024

Contents

1 Introduction

Concurrent and distributed algorithms are to be found in basically every modern software, and their correctness notably relies on the communication between cores and threads. In this work, we focus on verifying reliable message-passing channels, that is to say channels which preserve message order and where messages do not get lost. An usual way of verifying communication through channels is to use session types, however, it is not obvious how to handle the abortion of a communication du to an error. There already exists ways to extend session types to catch this behaviour[1], and the main focus during this internship was to find a way to extend the Actris logic[2] in a similar manner.

The results presented are formalized in Coq using the Iris separation logic framework[3][4], and can be found on Github. The code implementation is done in a basic untyped ML-like language and could easily be transported to an existing language, like OCaml. The specifications are made compatible with the automated verification tool Diaframe[5], and can be used automatically.

The verification done in this project consists only in showing that the written programs don't get stuck. The results are expressed in terms of Hoare triplets $\{P\}\,\mathsf{prog}\,\{v.\ Q\ v\}$, meaning that under the precondition P, the program prog does never get stuck. Moreover, if it terminates, it returns a value v satisfying the postcondition $Q\ v$. If Q does not depend on the returned value, we write $\{P\}\,\mathsf{prog}\,\{Q\}$ for more convenience. Notice that nothing guarantees termination; we would thus have the perfectly valid Hoare triple $\{\mathsf{True}\}\,\mathsf{loop}\,()\,\{\mathsf{False}\}$ if loop only performs an infinite loop.

Reasonning about termination in concurrent programs is hard and is not a topic treated during this intership, although fair termination and deadlock freedom has been addressed in the litterature [6][7].

1.1 Separation Logic

The main particularity of separation logic is that propositions are seen as a multiset of resources that one can consume. Once these propositions are used, we do not have access to them anymore.

Concretely, we use the connectors $*$ and $-\!\!*$, $P -\!\!* Q$ meaning that there are enough resources for Q in P, and $P * Q$ meaning that we have both the resources of P and of Q. One can think of $*$ as $\wedge$ and of $-\!\!*$ as $\rightarrow$, but there are major differences. Most notably, it is not true in general that $P -\!\!* P*P$, which may seem a little confusing in the beginning. There even exists some propositions such that $(P*P) -\!\!* \mathsf{False}$, but where $P -\!\!* \mathsf{False}$ does not hold. This kind of proposition is especially important because it grants exclusivity: if at some point I have this proposition in one of the threads of my distributed programs, I can make sure that no other thread has it at the same point in time.

One key example of exclusive propositions is ownership. Everytime a pointer is allocated, we obtain a location l and a token $l \mapsto v$ inside the logic, where v is the value to which l is pointing:

$$\textsc{Alloc}\quad \frac{v \text{ is a value}}{\{\mathsf{True}\}\,\mathsf{ref}\ v\,\{l,\ l \mapsto v\}}$$

This proposition can then be used to read and write from the pointer. Writing requires full ownership of the pointer, meaning that it is only possible by providing the $l \mapsto v$ token. This ensures that no program is able to write to a location if it is not supposed to.

This notion can be extended to ghost variables. The point is that at any moment, a ghost variable can be allocated with a value v, which gives a proposition of the form $\exists\gamma,\ \gamma \hookrightarrow v$. Its ownership can be split between different threads, giving partial ownership which we write $\gamma \overset{q}{\hookrightarrow} v$, where $q \in \mathbb{Q} \cap]0; 1]$, but the value it is pointing to can only be modified with the full ownership. This is extremely useful to catch the behaviour of a program that doesn't explicitly allocate pointers corresponding to its logical state.

To use knoledge about what each thread is doing, we use invariants. These are duplicable (also called persistent) propositions, that is, such that $P -\!\!* P * P$, and can be initialized, opened and

closed. When we initialize or close an invariant, we have to provide resources. When we open it, we obtain its resources, but we will have to close it later. The invariants we use are atomic, which intuitively means that we have to close them one computation step after having opened them. This is necessary since each thread could access to some global variable and change the state of the whole system in one step.

Some programs however behave as though they exectuted in only one step, because they use their precondition for one instruction only, and then perform operations that do not affect the other threads. When this is the case, an invariant may be opened around the whole program, and we write its specification $\langle P \rangle$ prog $\langle v.\ Q\ v \rangle$. Most often however, such programs require some logical environment that is not shared with other threads in order to work correctly. We then write $\langle P_{public} | P_{private} \rangle$ prog $\langle v.\ Q_{public}\ v | Q_{private}\ v \rangle$ in order to distinguish which part of the pre - and postcondition come from and return to an invariant (the public part), and which part remains exclusive to the thread (the private part).

1.2 Session Types and Dependent Protocols

Basic session types can be thought of as types for communications and are defined by the following grammar:

$$S \ := \ !\tau.S \mid ?\tau.S \mid \mathsf{end}$$

With the idea that an endpoint with type $!\tau.S$ (respectively $?\tau.S$) will need to send (respectively receive) a message of type τ and then perform the actions imposed by S. We also define the dual $\overline{S}$ of S by $\overline{!\tau.S} := ?\tau.\overline{S}$, $\overline{?\tau.S} := !\tau.\overline{S}$ and $\overline{\mathsf{end}} := \mathsf{end}$. Note that as one could expect, the dual operation is involutive: $\overline{\overline{prot}} = prot$. Let us look at an example (Fig. 1) to understand the use and the limitations of session types defined like this. The code relies on some basic message passing functions new_chan, send and recv. new_chan creates two endpoints that can safely communicate over threads, send asynchronously sends a message from one endpoint to the other and recv blocks until the other party has sent a message and returns it.

```
let (c, c') = new_chan () in

send c 41;                          let x = recv c' in
let result = recv c in      ||      let y = x + 1
print_int result            ||      send c' y
```

Figure 1: Illustrating Communication through Channels

In this example, the thread on the left, which can be seen as a client, sends some data to the thread on the right, the server. The processed data is then sent back to the client who may work with it. A session type for this communication could be $!\mathbb{N}.?\mathbb{N}.\mathsf{end}$ on the client side and its dual on the server side. This brings multiple befefits. The system will not reach a state where both sides are waiting for each other to send a message, generating a deadlock. Furthermore, receiving a message comes with a guarantee on its type. This allows for example the client to perform operations on integers only upon receiving the processed data.

However, when doing program verification, one may want more information. For instance, it could be useful for the client to know that the value of the result is going to be 42, or more generally the value of whatever has been sent incremented by one. The sent information could even be more complicated to obtain, such as ownership of a pointer (Fig. 2). These issues are addressed by the Actris logic[2] by introducing dependent separation protocols, a generalization of session types to the Iris logic.

Dependent separation protocols can be defined by :

$$prot \ := \ \ !\,\vec{x} : \vec{\tau}\,\langle v \rangle\,\{P\}.prot \mid \ \ ?\,\vec{x} : \vec{\tau}\,\langle v \rangle\,\{P\}.prot \mid \mu\, rec.prot \mid \mathsf{end}$$

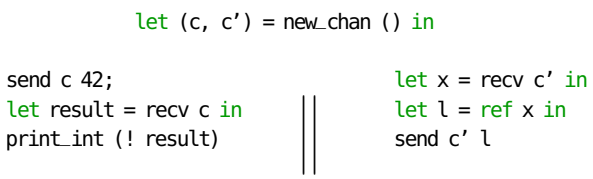


Figure 2: Sending a Pointer

$!\vec{x} : \vec{\tau}\,\langle v\rangle\,\{P\}.prot$ is a protocol that requires the endpoint to provide some values $\vec{x}$ of types $\vec{\tau}$ and a proposition P, after which the message v must be sent and the communication continues with the protocol $prot$. The protocols are said to be dependent, which means that v, P and especially $prot$ may depend on the provided variables. The protocol may thus completely change depending on the sent value. We omit the propositions when it is True, and write $!\vec{x} : \vec{\tau}\,\langle v\rangle\,.prot$ instead. The recursor $\mu\, rec.prot$ allows recursion inside protocols and can be rewritten as $prot[\mu\, rec.prot/rec]$, which is essential for algorithms that loop doing the same actions, like servers.

One protocol for the client side of the program 1 could for example be $!\,\langle 41\rangle\,.\ ?\,\langle 42\rangle\,.\mathsf{end}$, or more generally $!n : \mathbb{N}\,\langle n\rangle\,.\ ?\,\langle n+1\rangle\,.\mathsf{end}$. This means that the client gets to choose the number that is sent, and already knows at that moment what will be sent back.

The example 2 is more interesting, since it is dealing with shared references. The point is that in order to access the value of the returned pointer l, the client needs a $l \mapsto v$ predicate for some value v. This can be obtained with the protocol $!n : \mathbb{N}\,\langle n\rangle\,.\ ?l : \mathsf{loc}\,\langle l\rangle\,\{l \mapsto n\}.\mathsf{end}$. Upon receiving, the client does not know what location they will receive, however they do have the information that it will be pointing to the sent value.

1.3 Contributions

The focus of this internship was to extend the Actris logic with dependent session protocols able to catch the interruption of a communication. We provided the following features:

- We extended the Actris protocols by adding a possibility of cancelling the communication – on purpose or non deterministically.
- We implemented and verified a small library for message passing with shared memory, with the possibility to drop one endpoint during the communication.
- We provided examples using the library and verified their soundness using the extended version of the protocols.

2 Exceptional Actris

We built an extention of dependent protocols on top of Actris that would allow one side to give up the communication while the other side is still able to perform actions.

2.1 Cancellation Protocols

Let us have a look at this extension before going into further detail:

$$
\begin{array}{lll}
prot \;:=\; ! \vec{x} : \vec{\tau} \langle v \rangle \{P\}.prot \mid \; ? \vec{x} : \vec{\tau} \langle v \rangle \{P\}.prot \mid \mathsf{end} \mid & & \text{Actris protocols} \\
\quad \mu rec.prot \mid & & \text{Recursion} \\
\quad !_Q \vec{x} : \vec{\tau} \langle v \rangle \{P\}.prot \mid \; ?_Q \vec{x} : \vec{\tau} \langle v \rangle \{P\}.prot \mid & & \text{Nondeterministic cancelling} \\
\quad \lightning_Q \mid \lightning_Q & & \text{Deterministic cancelling}
\end{array}
$$

The idea of $\lightning_Q$ is the same than end for the side that is going to interrupt the communication, in so far as it is not possible to come back to a protocol allowing to send or receive information after reaching $\lightning_Q$. The main difference however is its dual $\overset{\lightning}{?}_Q$, which still allows to behave normally even if the other side has interrupted the connection. The nondeterministic version of the protocols allow to abort the communication instead of sending a message, which leads to a nondeterministic disjunction at the moment of receiving. The proposition Q is information sent at the moment of cancelling, which can possibly be used to rule out one branch in the disjunction. For example, $?_{\mathsf{False}} \vec{x} : \vec{\tau} \langle v \rangle \{P\}.prot$ is basically the same than $? \vec{x} : \vec{\tau} \langle v \rangle \{P\}.prot$, because one would need to produce False before being able to use the resources allowing to cancel. To have a better understanding of these protocols, let us have a look at the notion of *protocol subtyping*. The relation $prot_1 \sqsubseteq prot_2$ (Fig. 3) comes with the intuitition that following $prot_2$ instead of $prot_1$ should not make a difference for the other side. In Actris, this relation essentially allows to send a message before receiving one, which should not be a problem whatsoever since sending does not require any information.

$$
\textsc{Subprot-Swap}\quad ? \langle v \rangle . ! \langle v' \rangle .prot \sqsubseteq \; ! \langle v' \rangle . ? \langle v \rangle .prot
$$

$$
\textsc{Subprot-Send}\quad \frac{prot_1 \sqsubseteq prot_2}{! \langle v \rangle .prot_1 \sqsubseteq \; ! \langle v \rangle .prot_2}
\qquad
\textsc{Subprot-Dual}\quad \frac{prot_1 \sqsubseteq prot_2}{\overline{prot_2} \sqsubseteq \overline{prot_1}}
$$

$$
\textsc{Subprot-Payload-Send}\quad \frac{prot_1 \sqsubseteq prot_2 * P}{! \langle v \rangle \{P\}.prot_1 \sqsubseteq \; ! \langle v \rangle .prot_2}
\qquad
\textsc{Subprot-Refl}\quad prot \sqsubseteq prot
$$

$$
\textsc{Subprot-Trans}\quad \frac{prot_1 \sqsubseteq prot_2 * prot_2 \sqsubseteq prot_3}{prot_1 \sqsubseteq prot_3}
$$

Figure 3: Protocol Subtyping Rules in Actris

Let us have a look at some subtyping rules for the introduced protocols (Fig. 4). When the communication is interrupted from the other side, it remains possible to send messages in the void, although they will never be received. This is illustrated by the subtyping rule Subprot-Zap. The two other rules allow to either send a message or abort the communication, without having to make a choice in advance. This is necessary in most cases, because the knowledge of when an error happened indide a program is almost never available at the initialization. On the other side, this means that something could be received or not with no information beforehand. $?_Q \vec{x} : \vec{\tau} \langle v \rangle \{P\}.prot$ is defined as $\overline{!_Q \vec{x} : \vec{\tau} \langle v \rangle \{P\}.\overline{prot}}$, which comes with the rules induced by Subprot-Dual.

The purpose of $\lightning_Q$ and $\overset{\lightning}{?}_Q$ is not to be explicitely mentionned in the protocol of a communication, although this is possible. Rather, they are supposed to be reached from the nondeterministic send and receive protocols using subtyping rules.

Subprot-Zap
$\lightning_Q \sqsubseteq\ !\vec{x} : \vec{\tau}\,\langle v\rangle\,\{P\}.\lightning_Q$

Subprot-CancelSend-Cancel
$!_Q\,\vec{x} : \vec{\tau}\,\langle v\rangle\,\{P\}.prot \sqsubseteq \lightning_Q$

Subprot-CancelSend-Send
$!_Q\,\vec{x} : \vec{\tau}\,\langle v\rangle\,\{P\}.prot \sqsubseteq\ !\vec{x} : \vec{\tau}\,\langle v\rangle\,\{P\}.prot$

Figure 4: Extended Protocol Subtyping Rules

2.2 Channel Specifications

We provide a set of functions that allow message passing, and where a communication can be cancelled. Its behaviour is inspired by Rust's channel library, where drop can be called on an endpoint, closing the communication and dropping all messages unreceived messages in the buffer. From that moment, an attempt to receive a message will no longer be blocking the thread, and will immediately return an error. We emulate the fact that messages have a defined drop function by passing a closure with the sent message. This closure will be called if drop would have been called on the sent object.

```
type endpoint

new_chan : () → endpoint * endpoint
send : endpoint → val → (() → ()) → ()
recv : endpoint → option val
cancel : endpoint → ()
```

We define the specification for a drop function drop as drop_spec drop, which can be obtained by providing a Hoare triple $\{\mathsf{True}\}\,\mathsf{drop}\,\{\mathsf{True}\}$. Let us have a look at the specifications for the library (Fig. 5).

New-Chan-Spec
$\{\mathsf{True}\}$
new_chan ()
$\{(c, c'),\ c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$

Send-Spec
$\{c \rightarrowtail\ !\vec{x} : \vec{\tau}\,\langle v\rangle\,\{P\}.prot * (P[\vec{y}/\vec{x}] \wedge (\mathsf{drop\_spec\ drop}))\}$
send c $v[\vec{y}/\vec{x}]$ drop
$\{c \rightarrowtail prot[\vec{y}/\vec{x}]\}$

Cancel-Spec
$\{c \rightarrowtail \lightning_Q * Q\}$
cancel c
$\{\mathsf{True}\}$

Cancel-Spec-End
$\{c \rightarrowtail \mathsf{end}\}$
cancel c
$\{\mathsf{True}\}$

Recv-Spec
$\{c \rightarrowtail\ ?\vec{x} : \vec{\tau}\,\langle v\rangle\,\{P\}.prot\}$
recv c
$\{w,\ \exists\vec{y} : \vec{\tau},\ w = \mathsf{Some}\ v[\vec{y}/\vec{x}] * c \rightarrowtail prot[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\}$

Recv-Spec-Fail
$\{c \rightarrowtail \lightning_Q\}$
recv c
$\{\mathsf{None},\ (c \rightarrowtail \lightning_Q \wedge Q)\}$

CancelRecv-Spec
$\{c \rightarrowtail\ ?_Q\,\vec{x} : \vec{\tau}\,\langle v\rangle\,\{P\}.prot\}$
recv c
$\left\{\begin{array}{l} w,\ (\exists\vec{y} : \vec{\tau},\ w = \mathsf{Some}\ v[\vec{y}/\vec{x}] * c \rightarrowtail prot[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]) \\ \quad \vee (w = \mathsf{None} * (c \rightarrowtail \lightning_Q \wedge Q)) \end{array}\right\}$

Figure 5: Channel Library Specifications

At the moment of creating a new channel, one has to choose a protocol that the communication will follow. The endpoints will have to behave dually to each other for the communication to be correct. Upon using Send-Spec, one has to provide resources that need to be sent for the case where everything goes well and the other party receives the information (P), but it is also necessary to provide resources that will be consumed by drop. However, only one of the two will be used;

the other side either does or does not receive the message. Therefore, we do not use a separating conjonction $*$ to combine P and drop_spec drop in the precondition for Send-Spec, instead we combine them with a logical and. This way, both resources are accessible, but only one of them can actually be used. This allows the user of the library to use P to provide drop_spec drop, which most often is necessary. The specification for $\lightning_Q$ is quite intuitive; do note that no information is returned in the post condition. Calling $\lightning_Q$ removes access to any kind of resource, which prevents any use of send or recv in the future. recv returns an option that can only be None if the other side has called cancel. CancelRecv-Spec contains a disjunction that cannot be predicted beforehand, corresponding to the fact that the nondeterministic send protocol allows to send or cancel without providing information about the choice made. However, if cancel has been called, we know that all future calls to recv will return None.

We also have the very useful rule Subprot-Pointsto, which allows to actually make use of the protocol subtyping relation defined before.

$$\frac{c \rightarrowtail prot_1 * prot_1 \sqsubseteq prot_2}{c \rightarrowtail prot_2}\ \textsc{Subprot-Pointsto}$$

Using this rule, we are able to deduce rules that seem to be missing, such as for example CancelSend-Spec-Send or CancelSend-Spec-Cancel, which allow to actually make use of the nondeterministic send protocol, using respectively the Subprot-CancelSend-Send and Subprot-CancelSend-Cancel rules (Fig. 6).

CancelSend-Spec-Send

$\{c \rightarrowtail\ !_Q\, \vec{x} : \vec{\tau}\, \langle v \rangle\, \{P\}.prot * (P[\vec{y}/\vec{x}] \wedge (\mathsf{drop\_spec\ drop}))\}$

send c $v[\vec{y}/\vec{x}]$ drop

$\{c \rightarrowtail prot[\vec{y}/\vec{x}]\}$

CancelSend-Spec-Cancel

$\{c \rightarrowtail\ !_Q\, \vec{x} : \vec{\tau}\, \langle v \rangle\, \{P\}.prot * Q\}$

cancel c

$\{\mathsf{True}\}$

Figure 6: Some Rules Deducible from Subprot-Pointsto

In order to have a better intuition of these specifications, let us look back at example 2, in a slightly more fancy way (Fig. 7). Suppose the side producing the pointer is first sending various sorts of other messages (in our case a boolean), which could lead to an issue forcing to abort the communication. We use the notation let Some x = y in e for pattern matching that produces an error if the None clause is reached.

```
let (c, c') = new_chan () in

send c 42 (fun () => ());
let Some b = recv c in
if b then
   let Some result = recv c in
   print_int (! result)
else
   cancel c
||
let Some x = recv c' in
send c (f x) (fun () => ());
let l = ref x in
send c' l (fun () => free l)
```

Figure 7: Sending a Pointer

Without any information about the boolean function f, one cannot predict whether the left hand side thread will behave normally or abruptly cancel the communication. Thus, we will not know if the program will print 42 or not, but we still want to prove it does not crash. To do so, we can use the protocol $!\, n : \mathbb{N}\, \langle n \rangle\, .\, ?\, b : \mathbb{B}\, \langle b \rangle\, .(\mathsf{if}\ b\ \mathsf{then}\ \ ?\, l : \mathsf{loc}\, \langle l \rangle\, \{l \mapsto n\}.\mathsf{end}\ \mathsf{else}\ \lightning_{\mathsf{True}})$. Let us go through the right thread of the program step by step:

$$\{c' \rightarrowtail\ ?\, n : \mathbb{N}\, \langle n \rangle\, .\, !\, b : \mathbb{B}\, \langle b \rangle\, .(\mathsf{if}\ b\ \mathsf{then}\ \ !\, l : \mathsf{loc}\, \langle l \rangle\, \{l \mapsto n\}.\mathsf{end}\ \mathsf{else}\ \lightning_{\mathsf{True}})\}$$
$$\mathsf{let\ Some}\ x = \mathsf{recv}\ c'\ \mathsf{in}$$
$$\{c' \rightarrowtail\ !\, b : \mathbb{B}\, \langle b \rangle\, .(\mathsf{if}\ b\ \mathsf{then}\ \ !\, l : \mathsf{loc}\, \langle l \rangle\, \{l \mapsto n\}.\mathsf{end}\ \mathsf{else}\ \lightning_{\mathsf{True}})\}$$
$$\mathsf{send}\ c'\ (f\ n)\ (\mathsf{fun}\ () \Rightarrow ());$$
$$\{c' \rightarrowtail \mathsf{if}\ f\ n\ \mathsf{then}\ \ !\, l : \mathsf{loc}\, \langle l \rangle\, \{l \mapsto n\}.\mathsf{end}\ \mathsf{else}\ \lightning_{\mathsf{True}}\}$$
$$\mathsf{let}\ l = \mathsf{ref}\ n\ \mathsf{in}$$
$$\{c' \rightarrowtail \mathsf{if}\ f\ n\ \mathsf{then}\ \ !\, l : \mathsf{loc}\, \langle l \rangle\, \{l \mapsto n\}.\mathsf{end}\ \mathsf{else}\ \lightning_{\mathsf{True}} * l \mapsto n\}$$

The last instruction remaining is the most tricky, for two reasons:

- With no further knowledge about f, we need to do a case disjunction on $f\ n$. One of the two branches corresponds to the send that is actually performed, the other corresponds to sending the location l through the channel that has been or will be cancelled. Basically, that means calling free on it.
- The current logical state of the program does not contain drop_spec (fun () ⇒ free l). This would correspond to $\{\mathsf{True}\}\,\mathsf{free}\ l\,\{\mathsf{True}\}$, which we cannot prove without any further information (the specification for free is $\{l \mapsto x\}\,\mathsf{free}\ l\,\{\mathsf{True}\}$). However, we do have $l \mapsto n \mathrel{-\!\!*} \{\mathsf{True}\}\,\mathsf{free}\ l\,\{\mathsf{True}\}$, which easily gives $l \mapsto n \mathrel{-\!\!*} (l \mapsto n \wedge \mathsf{drop\_spec}\ (\mathsf{fun}\ () \Rightarrow \mathsf{free}\ l))$.

The simplest way to deal with the branching in the protocol is probably to note that $\lightning_{\mathsf{True}} \sqsubseteq\ !\, l : \mathsf{loc}\, \langle l \rangle\, \{l \mapsto n\}.\lightning_{\mathsf{True}}$, which gives the nice following result:

$$(\mathsf{if}\ f\ n\ \mathsf{then}\ \ !\, l : \mathsf{loc}\, \langle l \rangle\, \{l \mapsto n\}.\mathsf{end}\ \mathsf{else}\ \lightning_{\mathsf{True}}) \sqsubseteq\ !\, l : \mathsf{loc}\, \langle l \rangle\, \{l \mapsto n\}.(\mathsf{if}\ f\ n\ \mathsf{then\ end\ else}\ \lightning_{\mathsf{True}})$$

Verifying the last instruction is now much easier:

$$\{c' \rightarrowtail\ !\, l : \mathsf{loc}\, \langle l \rangle\, \{l \mapsto n\}.(\mathsf{if}\ f\ n\ \mathsf{then\ end\ else}\ \lightning_{\mathsf{True}}) * (l \mapsto n \wedge \mathsf{drop\_spec}\ (\mathsf{fun}\ () \Rightarrow \mathsf{free}\ l))\}$$
$$\mathsf{send}\ c'\ l\ (\mathsf{fun}\ () \Rightarrow \mathsf{free}\ l);$$
$$\{c' \rightarrowtail \mathsf{if}\ f\ n\ \mathsf{then\ end\ else}\ \lightning_{\mathsf{True}}\}$$

3 Implementation

In order to better understand what kind of program we are dealing with, let us have a look at a possibility to implement the library described. The implementation is designed to be efficient (even though it is not highly optimised), and is entirely mechanised in Coq.

The major part of the work on implementing channels was to implement queues, with the following signatures:

```
type sender, receiver

new_queue : () → (sender * receiver)
enqueue : sender → val → (() → ()) → ()
dequeue : receiver → option val
cancel_sender : sender → ()
cancel_receiver : receiver → ()
```

We then define the channels using two queues:

```
let endpoint = sender * receiver

let new_chan () =
    let (s, r) = new_queue () in
    let (s', r') = new_queue () in
    ((s, r'), (s', r))

let send (s, r) v drop =
    enqueue s v drop

let recv (s, r) =
    dequeue r

let cancel (s, r) =
    cancel_sender s;
    cancel_receiver r
```

The interesting part of the implementation is of course hidden in the queue library.

3.1 Single Producer Single Consumer Queues

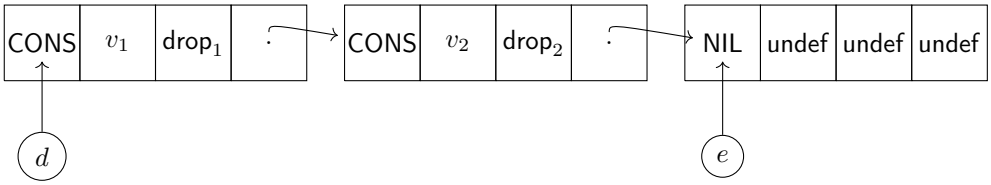
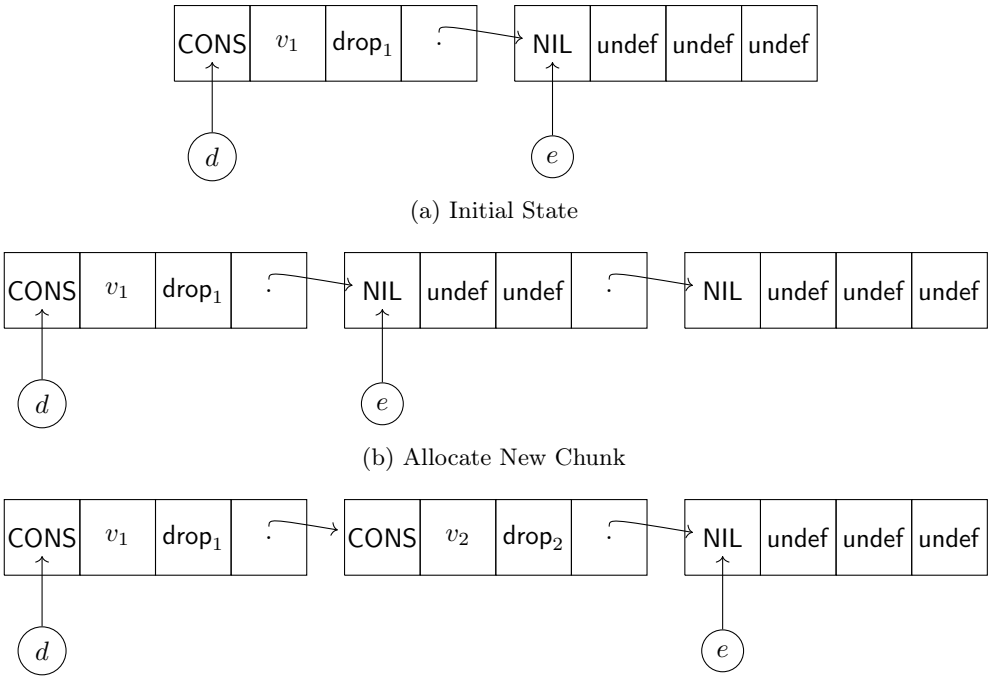


Figure 8: A Single Producer Single Consumer Queue

We implement a queue by generating a pair of pointers (e, d); d pointing to the front of the queue, e pointing to the back (Fig. 8). Whenever a new element is to be enqueued, a new chunk of memory would be allocated, and tagged by NIL, meaning that it is the new end of the queue. Then, the current end will be filled with the provided information, after which the tag can safely be changed to CONS, meaning that the last element of the chunk is pointing to the rest of the queue (Fig. 9).

The dequeue operation will never read information from chunks that are not tagged CONS. Calling cancel_sender, will only change the tag of the back from NIL to CANCEL. Semantically, the difference is that when trying to dequeue an element, dequeue will return None if the back flag is set to CANCEL, whereas it will loop waiting for the sender if it is set to NIL. cancel_receiver is a bit more complicated, as it will go through the entire queue, call each drop, and switch the last tag to CANCEL once it arrives at the end. Changing a flag has to be done very carefully, since both sides



(a) Initial State

(b) Allocate New Chunk

(c) Store Data and Change Tag

Figure 9: send Operation

of the queue could potentially try to access it at the same time, which may lead to misbehaviours. In order to prevent this from happening, we use the function CmpXchg (Compare and Exchange), which allows to atomically change the value of a location, based on a test on its previous value (Fig. 10).

$$\textsc{CmpXchg-Succ}\quad \langle l \mapsto v_1 \rangle \;\; \mathsf{CmpXchg}\ l\ v_1\ v_2 \;\; \langle (v_1, \mathsf{true}),\ l \mapsto v_2 \rangle$$

$$\textsc{CmpXchg-Fail}\quad \frac{v_1 \neq v_1'}{\langle l \mapsto v_1' \rangle \;\; \mathsf{CmpXchg}\ l\ v_1\ v_2 \;\; \langle (v_1', \mathsf{false}),\ l \mapsto v_1' \rangle}$$

Figure 10: Rules for CmpXchg

The moment when this function is called in the code (Fig. 11) is usually the most critical in the function, and the only moment when a bug due to concurrency could occur. Note that if multiple threads where allowed to concurrently share one side of the queue, a lot of problems would immediately happen. For example, enqueue relies on the fact that the part of the queue it is pointing to is the last part, that is to say, never tagged by CONS. On the other side, dequeue could generate a lot of errors if called concurrently on the same pointer, since one call would free memory that could be accessed by the other threads. It is therefore necessary that the specifications reflect the fact that an endpoint cannot be used by two threads at the same time.

3.2 Queue Specifications

Let us have a look at the specifications provided for the queue library (Fig. 12). Some technical details have been omitted because they do not bring useful insights and make the rules unnecessarily

```
let sender = ref int
let receiver = ref int
let CANCEL = 0
let NIL = 1
let CONS = 2

let alloc_nil () =
(* A chunk of size 4 initialized with () *)
    let l = array_alloc 4 () in
    l ← NIL;
    l

let enqueue e v drop =
    let l = alloc_nil () in
    let current = !e in
    (current + 1) ← v;
    (current + 2) ← drop;
    (current + 3) ← l;
    let (_, b) = CmpXchg current NIL CONS in
    if b then e ← l else
    drop ();
    array_free 4 l;

let cancel_sender e =
    let current = !e in
    free e;
    let (_, b) = CmpXchg current NIL CANCEL in
    if b then () else
    array_free 4 current
```

```
let new_queue () =
    let l = alloc_nil () in
    (ref l, ref l)

let rec dequeue d =
    let last = !d in
    let tag = !last in
    if tag = NIL then dequeue d else
    if tag = CANCEL then None else
    let next = !(last + 3) in
    d ← next;
    let v = !(last + 1) in
    array_free 4 last;
    Some v

let rec cancel_receiver d =
    let last = !d in
    let (tag, _) = CmpXchg last NIL CANCEL in
    if tag = CONS then
        let drop = !(last + 2) in
        let next = !(last + 3) in
        d ← next;
        drop ();
        array_free 4 last;
        cancel_receiver d
    else
    free d;
    if tag = NIL then () else
    array_free 4 last
```

Figure 11: Single Producer Single Consumer Queue Library

harder to understand, although they are essential to prove soundness of the global results. We introduce the following notations :

- $\mathsf{E}\ e$ is a proposition standing for enqueue ownership of the queue. A thread owning this predicate is able to use specifications related to enqueue with the endpoint e.
- Similarily, a thread owning $\mathsf{D}\ d$ is allowed to call dequeue d safely.
- $\mathsf{Q}_{b_e}^{b_d}\ e\ d\ q$ is the resource corresponding to the current logical state of the queue. b_e (respectively b_d) is a boolean reflecting whether or not the back (respectively the front) of the queue has been cancelled. q is a list containing pairs (v, drop), where v is a message sent with cancel function drop. This list morally reflects the messages that are in the current buffer, waiting to be received by dequeue d. If cancel_receiver d has already been called, that is to say if $b_d = \mathsf{false}$, q loses its meaning and can be any list, which makes the specifications a bit easier to read.

Those propositions are related to another in the sense that can deduce from $\mathsf{E}\ e$ and $\mathsf{Q}_{b_e}^{b_d}\ e\ d\ q$ that $b_e = \mathsf{true}$. Similarily, owning $\mathsf{D}\ d$ gives that $b_d = \mathsf{true}$. This is because ownership of an

endpoint has to be given up in order to cancel the channel, which would cause a contradiction if the corresponding boolean was false. We also define the logically cancelled tokens $\mathsf{E}_C\ e$ and $\mathsf{D}_C\ d$, which have the meaning that even though no function has been called yet, the endpoint has already given up all of the resources allowing to interract with the queue, and the only possibility for it is to be cancelled. The existence of these tokens allows to perform logical actions ahead of time, which is needed to prove an atomic behaviour of cancel, even though it calls two functions which each require the same public resource $\mathsf{Q}_{b_e}^{b_d}\ e\ d\ q$. The rule allowing to give up $\mathsf{E}\ e$ in order to get back $\mathsf{E}_C\ e$ looks as follows:

$$\textsc{Logically-Cancel-Sender}\quad \frac{\mathsf{E}\ e * \mathsf{Q}_{b_e}^{b_d}\ e\ d\ q}{\mathsf{E}_C\ e * \mathsf{Q}_{\mathsf{false}}^{b_d}\ e\ d\ q}$$

The rule allowing to get $\mathsf{D}_C\ d$ is much more complicated because all of the $\mathsf{drop\_spec}$ drop for drop in q need to be provided, which may require to open an invariant thus needs more generalization. The idea behind it remains the same.

$\textsc{New-Queue-Spec}$
$\{\mathsf{True}\}$
$\quad \mathsf{new\_queue}\ ()$
$\{(e, d),\ \mathsf{E}\ e * \mathsf{D}\ d * \mathsf{Q}_{\mathsf{true}}^{\mathsf{true}}\ e\ d\ [\,]\}$

$\textsc{Enqueue-Spec}$
$\langle \mathsf{Q}_{b_e}^{b_d}\ e\ d\ q * \mathsf{if}\ b_d\ \mathsf{then\ True\ else\ drop\_spec\ drop} | \mathsf{E}\ e \rangle$
$\quad \mathsf{enqueue}\ e\ v\ \mathsf{drop}$
$\langle \mathsf{Q}_{\mathsf{true}}^{b_d}\ e\ d\ q +\!\!+ [(v, \mathsf{drop})] | \mathsf{E}\ e \rangle$

$\textsc{Dequeue-Spec}$
$\langle \mathsf{Q}_{b_e}^{b_d}\ e\ d\ q | \mathsf{D}\ d \rangle$
$\quad \mathsf{dequeue}\ d$
$\left\langle w,\ (\exists v\ \mathsf{drop}\ q',\ q = (v, \mathsf{drop}) :: q' * w = \mathsf{Some}\ v * \mathsf{Q}_{b_e}^{\mathsf{true}}\ e\ d\ q') \right.$
$\left. \vee\ (q = [\,] * w = \mathsf{None} * b_e = \mathsf{false} * \mathsf{Q}_{\mathsf{false}}^{\mathsf{true}}\ e\ d\ [\,]) \right| \mathsf{D}\ d \rangle$

$\textsc{Cancel-Sender-Spec}$
$\{\mathsf{E}_C\ e\}$
$\quad \mathsf{cancel\_sender}\ e$
$\{\mathsf{True}\}$

$\textsc{Cancel-Receiver-Spec}$
$\{\mathsf{D}_C\ d\}$
$\quad \mathsf{cancel\_receiver}\ d$
$\{\mathsf{True}\}$

Figure 12: Specifications for the Queue Library

One may be wondering about the fact that in $\textsc{Dequeue-Spec}$ when $\mathsf{dequeue}$ returns None, is possible to know that $b_e = \mathsf{false}$, which corresponds to the back of the queue being tagged with CANCEL. This comes from the fact that we are not providing any result in case of non-termination, which would be what happens if the queue is empty but not cancelled. This equality is much needed in practice since it gives the possibility to rule out the case where the function returns None when one knows that the sending part has not been cancelled yet.

Defining the provided predicates requires quite some effort, but there are three key components:

- The private tokens $\mathsf{E}\ e$ and $\mathsf{D}\ d$ contain the information that should never be used by the other side, such as $e \mapsto \mathit{current}$ for some location $\mathit{current}$, or ownership of the undefined part of $\mathit{current}$ which looks like $\exists v_1, (\mathit{current} + 1) \mapsto v_1$. Most notably, these do not contain any information about the tag of $\mathit{current}$, which is stored in the first part of the chunk.
- An invariant for the whole queue which contains ownership of the critical part of the queue, that is, the tag of the back cell. This invariant also contains ghost variables that can be synchronized with the private and public tokens in order to establish facts about the state of

this cell. Another key role it plays is to store all of the information inside the queue, which is provided when enqueue is called and claimed by dequeue.

- The public token $\mathsf{Q}_{b_e}^{b_d}\ e\ d\ q$ only contains ghost information about its arguments, and is needed to obtain data from the invariant.

3.3 Channels and Protocols

Once the bases are set, defining channels using two queues is straightforward. We provide intermediate specifications before adding protocols, in order to obtain simpler definitions and proofs (Fig. 13). The tokens used in the specifications are defined as one could expect:

$$\mathsf{C}\ c \qquad := \exists e\ d,\ c = (e,d) * \mathsf{E}\ e * \mathsf{D}\ d$$
$$\mathsf{Chan}_{b_c}^{b_{c'}}\ c\ c'\ q\ q' := \exists e\ d\ e'\ d',\ c = (e,d) * c' = (e',d') * \mathsf{Q}_{b_c}^{b_{c'}}\ e\ d'\ q * \mathsf{Q}_{b_{c'}}^{b_c}\ e'\ d\ q'$$

NEW-CHAN-SPEC'
$\{\mathsf{True}\}$
new_chan ()
$\{(c,c'),\ \mathsf{C}\ c * \mathsf{C}\ c' * \mathsf{Chan}_{\mathsf{true}}^{\mathsf{true}}\ c\ c'\ []\ []\}$

SEND-SPEC'
$\left\langle \mathsf{Chan}_{b_c}^{b_{c'}}\ c\ c'\ q\ q' * \mathsf{if}\ b_{c'}\ \mathsf{then\ True\ else\ drop\_spec\ drop} | \mathsf{C}\ c \right\rangle$
send c v drop
$\left\langle \mathsf{Chan}_{\mathsf{true}}^{b_{c'}}\ c\ c'\ q +\!\!+ [(v,\mathsf{drop})]\ q' | \mathsf{C}\ c \right\rangle$

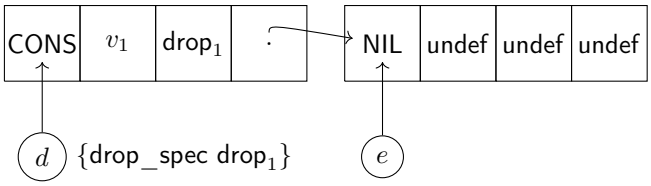
RECV-SPEC'
$\left\langle \mathsf{Chan}_{b_c}^{b_{c'}}\ c\ c'\ q\ q' | \mathsf{C}\ c \right\rangle$
recv c
$\left\langle w,\ (\exists v\ \mathsf{drop}\ q_1',\ q' = (v,\mathsf{drop}) :: q_1' * w = \mathsf{Some}\ v * \mathsf{Chan}_{\mathsf{true}}^{b_{c'}}\ c\ c'\ q\ q_1') \right.$
$\left. \vee\ (q' = [] * w = \mathsf{None} * b_{c'} = \mathsf{false} * \mathsf{Chan}_{\mathsf{true}}^{\mathsf{false}}\ c\ c'\ q\ []) \middle| \mathsf{C}\ c \right\rangle$

CANCEL-SPEC'
$\left\langle \mathsf{Chan}_{b_c}^{b_c c'}\ c\ c'\ q\ q' * \mathop{\Large\ast}_{(\_,\mathsf{drop})\in q'} \mathsf{drop\_spec\ drop} \middle| \mathsf{C}\ c \right\rangle$
cancel c
$\left\langle \mathsf{Chan}_{\mathsf{false}}^{b_{c'}}\ c\ c'\ q\ [] \right\rangle$

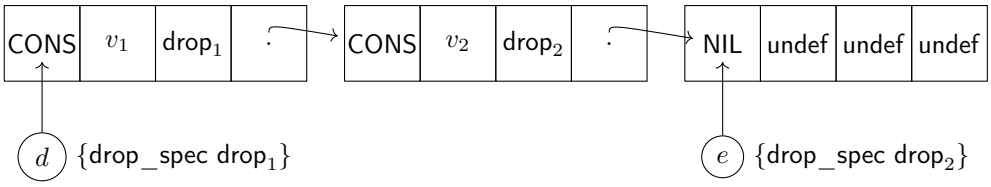
Figure 13: Atomic Channel Specifications

CANCEL-SPEC' is the only specification that looks new in so far as it requires all of the resources to every drop in the current buffer. These resources however could not be enough in the case where send is called after the cancel has started, but before it has effectively changed the queue's tag to CANCEL. In this case, send will need to provide the resources itself, allowing cancel to access them later on (Fig. 14). In order to make this work in practice, both sides put the cancelling resources inside an the queue invariant, so that cancel_receiver has access to it when needed.

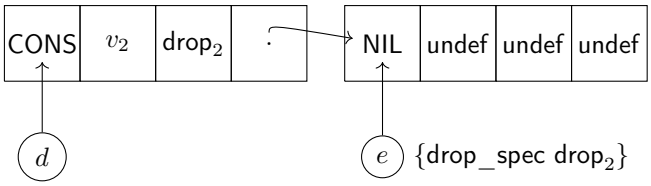
In order to move to the next layer, we define the cancelling protocols by lifting the messages to options and adding information that will be hidden to the user but is needed for verification purposes. A None message will be sent upon calling cancel, all other messages are of the form Some



(a) cancel_receiver Called



(b) In-Between enqueue



(c) Cancelling Goes On

Figure 14: send Operation

(v, drop). In the definitions, use the notation

$$\langle \tilde{v} \rangle \{P\}.prot := \exists \mathsf{drop}\, \gamma,\ \langle \mathsf{Some}\ (v, \mathsf{drop}, \gamma) \rangle \{(P \wedge \mathsf{drop\_spec}\ \mathsf{drop}) \vee \gamma \overset{1/2}{\hookrightarrow} \mathsf{false}\}.prot$$

and the tilde will later be omitted when there is no ambiguity. γ is a ghost variable tracking whether the side receiving the message has been cancelled. Therefore, the disjunction in the sent information makes it easier for the sender to send a message, although the receiving side will always be able to rule out the right side of the disjunction. Furthermore, in our setup, once γ has been set to false, it is not allowed to come back to true, which makes $\gamma \overset{1/2}{\hookrightarrow} \mathsf{false}$ persistent.

$$\begin{aligned}
\lightning_Q &:= !\ \langle \mathsf{None} \rangle\, \{Q\}.\mu\, rec.\ ?\, v : \mathsf{val}\, \langle \tilde{v} \rangle\, .rec \\
!_Q\, m &:= !\, b : \mathbb{B}\, (\mathsf{if}\ b\ \mathsf{then}\ m\ \mathsf{else}\ \langle \mathsf{None} \rangle \{Q\}.\mu\, rec.\ ?\, v : \mathsf{val}\, \langle \tilde{v} \rangle\, .rec) \\
\overline{\lightning}_Q &:= \overline{\lightning_Q} \\
?_Q\, m &:= \overline{!_Q\, \overline{m}}
\end{aligned}$$

By using these definitions, the rules about subtyping (Fig. 4) can be shown quite easily. Subprot-Zap is a consequence of the swapping rule Subprot-Swap which allows to send a message before receiving None. Subprot-CancelSend-Send and Subprot-CancelSend-Cancel can respectively be obtained by providing the booleans true and false. We define the connective $c \rightarrowtail prot$ using the Actris predicates for protocol ownership $\gamma_p \hookrightarrow_\bullet prot$ and $(\gamma_p, \gamma_p') \hookrightarrow_\circ (q, q')$, which enable a link between protocols and the logical queue states (Fig. 15). The notation $\boxed{Inv}$ corresponds to an invariant Inv of the program.

$$\begin{aligned}
\mathsf{lift}\ \gamma\ q\ b &:= [\mathsf{Some}\ (v, \mathsf{drop}, \gamma) | (v, \mathsf{drop}) \in q] +\!\!+ \mathsf{if}\ b\ \mathsf{then}\ []\ \mathsf{else}\ [\mathsf{None}] \\
Inv\ \gamma\ \gamma'\ c\ c'\ \gamma_p\ \gamma'_p &:= \exists q\ q'\ b_c\ b_{c'},\ \gamma \overset{1/2}{\hookrightarrow} b_c * \gamma' \overset{1/2}{\hookrightarrow} b_{c'} * \\
&\quad \mathsf{Chan}^{b_{c'}}_{b_c}\ c\ c'\ q\ q' * (\gamma_p, \gamma'_p) \hookrightarrow_\circ (\mathsf{lift}\ \gamma\ q\ b_c, \mathsf{lift}\ \gamma'\ q'\ b_{c'}) \\
c \rightarrowtail prot &:= \exists \gamma\ \gamma'\ c'\ \gamma_p\ \gamma'_p,\ \gamma \overset{1/2}{\hookrightarrow} \mathsf{true} * \mathsf{C}\ c * \gamma_p \hookrightarrow_\bullet prot * \boxed{Inv\ \gamma\ \gamma'\ c\ c'\ \gamma_p\ \gamma'_p}
\end{aligned}$$

Prot-Own-Alloc
$$\exists \gamma_p\ \gamma'_p,\ \gamma_p \hookrightarrow_\bullet prot * \gamma'_p \hookrightarrow_\bullet \overline{prot} * (\gamma_p, \gamma'_p) \hookrightarrow_\circ ([], [])$$

Prot-Own-Subprot
$$\frac{prot_1 \sqsubseteq prot_2 * \gamma_p \hookrightarrow_\bullet prot_1}{\gamma_p \hookrightarrow_\bullet prot_2}$$

Prot-Own-Sym
$$\frac{(\gamma_p, \gamma'_p) \hookrightarrow_\circ (q, q')}{(\gamma'_p, \gamma_p) \hookrightarrow_\circ (q', q)}$$

Prot-Own-Send
$$\frac{\gamma_p \hookrightarrow_\bullet\ !\,\vec{x} : \vec{\tau}\,\langle v\rangle\,\{P\}.prot * (\gamma_p, \gamma'_p) \hookrightarrow_\circ (q, q') * P[\vec{y}/\vec{x}]}{\gamma_p \hookrightarrow_\bullet prot[\vec{y}/\vec{x}] * (\gamma_p, \gamma'_p) \hookrightarrow_\circ (q +\!\!+ [v[\vec{y}/\vec{x}]], q')}$$

Prot-Own-Recv
$$\frac{\gamma_p \hookrightarrow_\bullet\ ?\,\vec{x} : \vec{\tau}\,\langle v\rangle\,\{P\}.prot * (\gamma_p, \gamma'_p) \hookrightarrow_\circ (q, w :: q')}{\exists \vec{y} : \vec{\tau},\ w = v[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}] * \gamma_p \hookrightarrow_\bullet prot[\vec{y}/\vec{x}] * (\gamma_p, \gamma'_p) \hookrightarrow_\circ (q, q')}$$

Figure 15: Protocol Ownership Rules in Actris

In order to have a better understanding of how this works, let us have a look at a proof for Send-Spec. We consider a version without parameters, which makes the notations lighter and does not remove the expressivity of the statement. We start with the precondition $c \rightarrowtail\ !\,\langle v\rangle\,\{P\}.prot * (P \wedge \mathsf{drop\_spec}\ \mathsf{drop})$, which can be unfolded as follows :

$$\gamma \overset{1/2}{\hookrightarrow} \mathsf{true} * \mathsf{C}\ c * \gamma_p \hookrightarrow_\bullet\ !\,\mathsf{drop}\,\gamma_1\,\langle (v, \mathsf{drop}, \gamma_1)\rangle\,\{(P \wedge \mathsf{drop\_spec}\ \mathsf{drop}) \vee \gamma_1 \overset{1/2}{\hookrightarrow} \mathsf{false}\}.prot * \boxed{Inv\ \gamma\ \gamma'\ c\ c'\ \gamma_p\ \gamma'_p} * (P \wedge \mathsf{drop\_spec}\ \mathsf{drop})$$

Since our goal is to prove that $\mathsf{send}\ c\ v\ \mathsf{drop}$ executes correctly, we can consume $\mathsf{C}\ c$ and apply Send-Spec' using the invariant, which gives two requirements: the public precondition $\mathsf{Chan}^{b_{c'}}_{b_c}\ c\ c'\ q\ q' * \mathsf{if}\ b_{c'}\ \mathsf{then}\ \mathsf{True}\ \mathsf{else}\ \mathsf{drop\_spec}\ \mathsf{drop}$ should be satisfied once the invariant opened, and the invariant should be closed again using the resources provided in the public postcondition $\mathsf{Chan}^{b_{c'}}_{\mathsf{true}}\ c\ c'\ q +\!\!+ [(v, \mathsf{drop})]\ q'$. Once the invariant opened, we have access to the following resources:

$$\begin{aligned}
&\gamma \overset{1/2}{\hookrightarrow} \mathsf{true} * \gamma_p \hookrightarrow_\bullet\ !\,\mathsf{drop}\,\gamma_1\,\langle (v, \mathsf{drop}, \gamma_1)\rangle\,\{(P \wedge \mathsf{drop_spec}\ \mathsf{drop}) \vee \gamma_1 \overset{1/2}{\hookrightarrow} \mathsf{false}\}.prot * \\
&\quad (P \wedge \mathsf{drop_spec}\ \mathsf{drop}) * \gamma \overset{1/2}{\hookrightarrow} b_c * \gamma' \overset{1/2}{\hookrightarrow} b_{c'} * \\
&\quad \mathsf{Chan}^{b_{c'}}_{b_c}\ c\ c'\ q\ q' * (\gamma_p, \gamma'_p) \hookrightarrow_\circ (\mathsf{lift}\ \gamma\ q\ b_c, \mathsf{lift}\ \gamma'\ q'\ b_{c'}) * \boxed{Inv\ \gamma\ \gamma'\ c\ c'\ \gamma_p\ \gamma'_p}
\end{aligned}$$

First of all, γ can only point to one variable at the time, which gives $b_c = \mathsf{true}$. It is also necessary to choose how to split the resources into the two different goals, which actually requires to know the value of $b_{c'}$. This can be solved by case disjunction. Of course, both cases work, but let us consider the case where the other side has been cancelled and where the message will never be read ($b_{c'} = \mathsf{false}$). Since P will never be received, the conjonction $(P \wedge \mathsf{drop\_spec}\ \mathsf{drop})$ can be simplified as $\mathsf{drop\_spec}\ \mathsf{drop}$. After all the simplifications have been done, the resources available look as follows:

$$\begin{aligned}
&\gamma \overset{1/2}{\hookrightarrow} \mathsf{true} * \gamma_p \hookrightarrow_\bullet\ !\,\mathsf{drop}\,\gamma_1\,\langle (v, \mathsf{drop}, \gamma_1)\rangle\,\{(P \wedge \mathsf{drop_spec}\ \mathsf{drop}) \vee \gamma_1 \overset{1/2}{\hookrightarrow} \mathsf{false}\}.prot * \\
&\quad \mathsf{drop_spec}\ \mathsf{drop} * \gamma \overset{1/2}{\hookrightarrow} \mathsf{true} * \gamma' \overset{1/2}{\hookrightarrow} \mathsf{false} * \mathsf{Chan}^{\mathsf{false}}_{\mathsf{true}}\ c\ c'\ q\ q' * \\
&\quad (\gamma_p, \gamma'_p) \hookrightarrow_\circ (\mathsf{lift}\ \gamma\ q\ \mathsf{true}, \mathsf{lift}\ \gamma'\ q'\ \mathsf{false}) * \boxed{Inv\ \gamma\ \gamma'\ c\ c'\ \gamma_p\ \gamma'_p}
\end{aligned}$$

Since $\gamma' \overset{1/2}{\hookrightarrow} \mathsf{false}$ is owned, Prot-Own-Send can be called, changing the ghost information about the protocols. $\gamma' \overset{1/2}{\hookrightarrow} \mathsf{false}$ being persistent, it can be used without giving it up. After using $\mathsf{drop\_spec}\ \mathsf{drop}$ and $\mathsf{Chan}^{\mathsf{false}}_{\mathsf{true}}\ c\ c'\ q\ q'$ for the public precondition and receiving $\mathsf{Chan}^{\mathsf{false}}_{\mathsf{true}}\ c\ c'\ q\ +\!\!+[(v, \mathsf{drop})]\ q'$ from the public postcondition, the invariant needs to be closed using the following resources:

$$\gamma \overset{1/2}{\hookrightarrow} \mathsf{true} * \gamma_p \hookrightarrow_\bullet prot * *\gamma \overset{1/2}{\hookrightarrow} \mathsf{true} * \gamma' \overset{1/2}{\hookrightarrow} \mathsf{false} * \mathsf{Chan}^{\mathsf{false}}_{\mathsf{true}}\ c\ c'\ q\ +\!\!+[(v, \mathsf{drop})]\ q' *$$
$$(\gamma_p, \gamma'_p) \hookrightarrow_\circ ((\mathsf{lift}\ \gamma\ q\ \mathsf{true})\ +\!\!+[(v, \mathsf{drop})], \mathsf{lift}\ \gamma'\ q'\ \mathsf{false}) * \boxed{Inv\ \gamma\ \gamma'\ c\ c'\ \gamma_p\ \gamma'_p}$$

By noticing that $(\mathsf{lift}\ \gamma\ q\ \mathsf{true})\ +\!\!+[(v, \mathsf{drop})] = \mathsf{lift}\ \gamma\ (q\ +\!\!+[(v, \mathsf{drop})])\ \mathsf{true}$, the invariant can be closed and the private postcondition $\mathsf{C}\ c$ obtained back. Now, the postcondition of Send-Spec needs to be satisfied, which corresponds to the goal

$$\frac{\gamma \overset{1/2}{\hookrightarrow} \mathsf{true} * \gamma_p \hookrightarrow_\bullet prot * \mathsf{C}\ c * \boxed{Inv\ \gamma\ \gamma'\ c\ c'\ \gamma_p\ \gamma'_p}}{c \rightarrowtail prot}$$

That can be proved by definition of $c \rightarrowtail prot$.

4 An Example: Merge Sorting

Let us now look at a possible way to use this library in practice. Suppose to be given an implementation of lists that can be modified in place, as well as some basic functions used to implement merge sort.

```
merge : ('a → 'a → option bool) → list 'a → list 'a → bool
split : list 'a → list 'a
length : list 'a → int
```

merge takes a comparison function as an argument as well as two sorted lists, and tries to merge the two lists into one sorted list. The sorted list is stored in the first argument. However, the comparison function may fail at comparing two elements, and return None. In this case, merge returns false and gives no guarantee on the lists being sorted. Otherwise, it returns true. split splits its argument in two, and returns one of the halves. The other half is still owned by the argument.

We give some specifications for those functions by linking the physical objects with logical lists (Fig. 16). These depend on some predicates, on particular ($\mathsf{cmp\_spec}_R\ cmp$), which has the meaning that cmp reproduces the behaviour of the partial order R. If cmp is called on two elements that are not in R, it will return None and provide the logical token cmp_fail.

Merge-Spec

$$\frac{(\mathsf{cmp\_spec}_R\ cmp) * \mathsf{sorted}_R\ l * \mathsf{sorted}_R\ l'}{\{\mathsf{is\_list}\ vs\ l * \mathsf{is\_list}\ vs'\ l'\}\quad \mathsf{merge}\ cmp\ vs\ vs'\quad \left\{b,\ \mathsf{if}\ b\ \mathsf{then}\ \begin{array}{l}\exists l_{merged},\ \mathsf{is\_list}\ vs\ l_{merged} * \\ \mathsf{sorted}_R\ l_{merged} * \mathsf{Permutation}\ (l +\!\!+ l')\ l_{merged}\end{array}\ \mathsf{else\ cmp\_fail}\right\}}$$

Split-Spec

$$\{\mathsf{is\_list}\ vs\ l\}\quad \mathsf{split}\ vs\quad \{vs',\ \exists l_1\ l_2,\ \mathsf{is\_list}\ vs\ l_1 * \mathsf{is\_list}\ vs'\ l_2 * l = l_1 +\!\!+ l_2\}$$

Length-Spec

$$\{\mathsf{is\_list}\ vs\ l\}\quad \mathsf{length}\ vs\quad \{(\mathsf{length}\ l),\ \mathsf{is\_list}\ vs\ l\}$$

Figure 16: Basic Utility Functions

Let us now look at a library concurrently sorting lists in place depending on an arbitrary comparison function (Fig. 17). Each recursive call of the sorting function sort_service forks two new threads, each of them sorting a different part of the list, then merges them together. If, at some point, the sorting of a list fails because of the compare function, the error will be propagated by cancelling channels, allowing the other part to stop blocking and to receive the information. If everything went well, unit is sent so that the other side stops blocking and is able to return.

One possible protocol for the sort_service could be:

$$\begin{aligned}\mathsf{sort\_prot}_R\ cmp := {} & ?\,(vs : \mathsf{loc})(l : \mathsf{list}\ \text{'a})\,\langle vs\rangle\,\{\mathsf{is\_list}\ vs\ l\}. \\ & !_{\mathsf{cmp\_fail}}\,(l' : \mathsf{list}\ \text{'a})\,\langle()\rangle\,\{\mathsf{is\_list}\ vs\ l' * \mathsf{sorted}_R\ l'\}.\mathsf{end}\end{aligned}$$

Which easily allows to prove some specification like:

Client-Spec

$$\frac{\mathsf{PartialOrder}\ R * \mathsf{cmp\_spec}_R\ cmp}{\{\mathsf{is\_list}\ vs\ l\}\quad \mathsf{client}\ vs\quad \{b,\ \mathsf{if}\ b\ \mathsf{then}\ \exists l',\ \mathsf{Permutation}\ l\ l' * \mathsf{sorted}_R\ l' * \mathsf{is\_list}\ vs\ l'\ \mathsf{else\ cmp\_fail}\}}$$

```
let rec sort_service cmp c =
    match recv c with
    |None => cancel c;
    |Some l =>
    if length l < 2 then
        (* List successfully sorted *)
        send c () unit_drop
    else
    let l' = split l in
    let (c1, c1') = new_chan () in
    let (c2, c2') = new_chan () in
    fork (sort_service cmp c1');
    fork (sort_service cmp c2');
    send c1 l (list_drop l);
    send c2 l' (list_drop l');
    match recv c1, recv c2 with
    | (), () => if merge l l' then
                    send c () unit_drop
                else
                    cancel c
    | _ => cancel c
```

```
let server c =
    match recv c with
    |None => cancel c
    |Some cmp => sort_service cmp c

let client cmp l =
    let (c, c') = new_chan () in
    fork (server c');
    send c cmp unit_drop;
    send c l (list_drop l);
    match recv c with
    (* Some elements in l can't be compared *)
    |None => false
    (* l is sorted in place *)
    |Some () => true
```

Figure 17: A Concurrent Sorting Library

Acknowledgments

This internship was made in compagny of Robbert Krebbers and the SwS team at the Radboud University Nijmegen. I would like to thank the team for their friendliness and their help when I needed it. I especially thank Robbert Krebbers and Ike Mulder for their help working with Iris, which is not always trivial.

References

[1] Simon Fowler, Sam Lindley, J Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[2] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019.

[3] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.

[4] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 205–217, 2017.

[5] Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification of fine-grained concurrent programs in iris. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 809–824, 2022.

[6] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. Trillium: Higher-order concurrent and distributed separation logic for intensional refinement. *Proceedings of the ACM on Programming Languages*, 8(POPL):241–272, 2024.

[7] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. Deadlock-free separation logic: Linearity yields progress for dependent higher-order message passing. *Proceedings of the ACM on Programming Languages*, 8(POPL):1385–1417, 2024.