Dependent Session Protocols in Separation Logic from First Principles

A Separation Logic Proof Pearl

Jules Jacobs
Radboud University

Jonas Kastberg Hinrichsen
Aarhus University

Robbert Krebbers
Radboud University

Message Passing Concurrency

```
c1,c2 = new_chan()

fork { c1.send(7); b = c1.recv(); ... }

fork { a = c2.recv(); c2.send(8); ... }
```

Actris (Hinrichsen, Bengtson, Krebbers):
Separation logic verification for message passing programs
(+ shared memory, locks, ...)

This work: MiniActris, a Proof Pearl version of Actris

```
c1,c2 = new_chan()
fork {
  s = ref(0)
  c1.send((100,s))
  for(i = 1..100) c1.send(i)
  c1.recv()
  assert(!s == 5050) // verify this
}
fork {
  n,s = c2.recv()
  for(i = 1..n) s ← c2.recv() + !s
  c2.send(())
}
```

```
c1,c2 = new_chan()
fork {
  s = ref(0)
  c1.send((100,s))
  for(i = 1..100) c1.send(i)
  c1.recv()
  assert(!s == 5050) // verify this
}
fork {
  n,s = c2.recv()
  for(i = 1..n) s ← c2.recv() + !s
  c2.send(())
}
```

Protocol:

$$
\begin{aligned}
c1 \rightarrowtail\ & \mathbf{!}\,(n : \mathbb{N}, s : \mathsf{Loc})\ \langle (n, s) \rangle \{ s \mapsto 0 \}. \\
& \mathbf{!}\,(i_1 : \mathbb{N})\ \langle i_1 \rangle \{\mathsf{True}\}. \ldots \mathbf{!}\,(i_n : \mathbb{N})\ \langle i_n \rangle \{\mathsf{True}\}. \\
& \mathbf{?} \langle () \rangle \{ s \mapsto \Sigma_1^n i_k \}. \,\mathbf{end}
\end{aligned}
$$

```
c1,c2 = new_chan()
fork {
  s = ref(0)
  c1.send((100,s))
  for(i = 1..100) c1.send(i)
  c1.recv()
  assert(!s == 5050) // verify this
}
fork {
  n,s = c2.recv()
  for(i = 1..n) s ← c2.recv() + !s
  c2.send(())
}
```

Protocol:

$$
\begin{aligned}
c2 \rightarrowtail\ & \mathbf{?}(n : \mathbb{N}, s : \mathsf{Loc})\ \langle (n, s) \rangle \{ s \mapsto 0 \}. \\
& \mathbf{?}(i_1 : \mathbb{N})\ \langle i_1 \rangle \{\mathsf{True}\}. \ldots \mathbf{?}(i_n : \mathbb{N})\ \langle i_n \rangle \{\mathsf{True}\}. \\
& \mathbf{!}\ \langle () \rangle \{ s \mapsto \Sigma_1^n i_k \}. \mathbf{end}
\end{aligned}
$$

MiniActris: a Proof Pearl version of Actris

Key idea: 3 layers:

1. single-shot channels (Dharda et al.)
2. functional session channels
3. imperative session channels

Key Iris ingredient: nested invariants

Layer 1: Single-shot channels

```
new1 () := ref None

send1 c v := c ← Some v

recv1 c := match !c with
           | Some v => free c; v
           | None => recv1 c
           end
```

Layer 1: Single-shot channels (specifications)

Protocols: $p \in \mathsf{Prot} \triangleq \{\mathsf{Send}, \mathsf{Recv}\} \times (\mathsf{Val} \to \mathsf{iProp})$

Dual protocol: $\overline{(\mathsf{Send}, \Phi)} \triangleq (\mathsf{Recv}, \Phi) \qquad \overline{(\mathsf{Recv}, \Phi)} \triangleq (\mathsf{Send}, \Phi)$

Channel points-to: $c \rightarrowtail^{\mathsf{base}} p \in \mathsf{iProp}$

Channel creation: $\{\mathsf{True}\}\ \mathbf{new1}\ ()\ \{c.\ c \rightarrowtail^{\mathsf{base}} p * c \rightarrowtail^{\mathsf{base}} \overline{p}\}$

Send message: $\{c \rightarrowtail^{\mathsf{base}} (\mathsf{Send}, \Phi) * \Phi\ v\}\ \mathbf{send1}\ c\ v\ \{\mathsf{True}\}$

Receive message: $\{c \rightarrowtail^{\mathsf{base}} (\mathsf{Recv}, \Phi)\}\ \mathbf{recv1}\ c\ \{v.\ \Phi\ v\}$

Layer 1: Single-shot channels (invariant)

$$\mathsf{tok}\,\gamma \triangleq \mathsf{own}\,\gamma\,(\mathsf{Excl}\,())$$

$$\mathsf{chan\_inv}\,\gamma_1\,\gamma_2\,\ell\,\Phi \triangleq (\underbrace{\ell \mapsto \texttt{None}}_{\text{(1) initial state}}) \lor (\underbrace{\exists v.\,\ell \mapsto \texttt{Some}\,v * \mathsf{tok}\,\gamma_1 * \Phi\,v}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\mathsf{tok}\,\gamma_1 * \mathsf{tok}\,\gamma_2}_{\text{(3) final state}})$$

$$c \xrightarrow{\text{base}} (tag, \Phi) \triangleq \exists \gamma_1, \gamma_2, \ell.\ \triangleright(c = \ell) * \boxed{\mathsf{chan\_inv}\,\gamma_1\,\gamma_2\,\ell\,\Phi} * \triangleright \begin{cases} \mathsf{tok}\,\gamma_1 & \text{if } tag = \mathsf{Send} \\ \mathsf{tok}\,\gamma_2 & \text{if } tag = \mathsf{Recv} \end{cases}$$

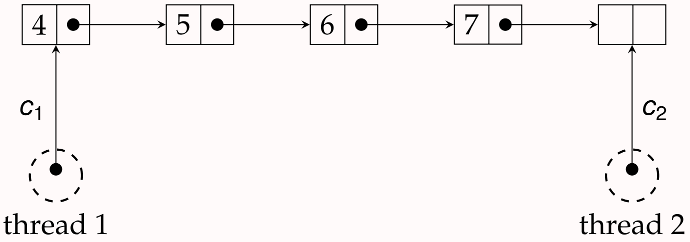
Layer 2: Functional session channels

(inspired by π-calculus – Kobayashi, Dharda)

```
new () := new1 ()

send c v := let c' = new1 () in send1 c (v,c'); c'

recv c := recv1 c
```



Layer 2: Functional session channels (specifications)

Key idea: define dependent session protocols as instances of single-shot protocols

$$\mathbf{!}\,(x:\tau)\,\langle v\rangle\{P\}.\,p \triangleq (\mathsf{Send}, \lambda(r:\mathsf{Val}).\,\exists(x:\tau), (c:\mathsf{Val}).\; r = (v\;x, c) * P\;x * c \xrightarrowtail{\mathsf{base}} p\;x)$$

$$\mathbf{?}(x:\tau)\,\langle v\rangle\{P\}.\,p \triangleq \overline{\mathbf{!}\,(x:\tau)\,\langle v\rangle\{P\}.\,\overline{p}}$$

Multi-step protocols require nested invariants:
Invariant of $c \xrightarrowtail{\mathsf{base}} \mathbf{!}\,(x:\tau)\,\langle v\rangle\{P\}.\,p$ contains $c' \xrightarrowtail{\mathsf{base}} p\;x$ for continuation channel.

Layer 3: Imperative channels

Functional channels are inconvenient:

```
let c' = send(c, 3) in
let (c'',v) = recv(c') in
...
```

We want:

```
c.send(3);
let v = c.recv() in
...
```

Imperative channels:

```
new_chan () := let c = new () in (ref c, ref c)

c.send(v) := c ← send (!c) v

c.recv() := let (v,c) = recv (!c) in c ← c'; v
```

Layer 3: Imperative channels (Actris-style specs)

Channel points-to:

$$c \xrightarrowtail{\text{imp}} p \triangleq \exists (\ell : Loc), (c' : Val).\, c = \ell * \ell \mapsto c' * c' \xrightarrowtail{\text{base}} p$$

Actris-style specs:

$$\{\text{True}\}\ \mathbf{new\_chan}\,()\ \{(c_1, c_2).\, c_1 \xrightarrowtail{\text{imp}} p * c_2 \xrightarrowtail{\text{imp}} \overline{p}\}$$

$$\{c \xrightarrowtail{\text{imp}} (\mathbf{!}\, x\, \langle v \rangle \{P\}.\, p) * P\ t\}\ c.\mathbf{send}(v\ y)\ \{c \xrightarrowtail{\text{imp}} (p\ y)\}$$

$$\{c \xrightarrowtail{\text{imp}} (\mathbf{?} x\, \langle v \rangle \{P\}.\, p)\}\ c.\mathbf{recv}()\ \{(v\ y).\, P\ y * c \xrightarrowtail{\text{imp}} (p\ y)\}$$

Guarded recursion

We want unbounded recursive protocols:

$$p \triangleq \mathbf{!}\,(x : \tau)\,\langle v \rangle\{Q\}.\,(\ldots p)$$

Already works!

Iris invariants are **contractive** $\implies$

$\mathbf{!}\,(x : \tau)\,\langle v \rangle\{P\}.\,p$ and $\mathbf{?}(x : \tau)\,\langle v \rangle\{P\}.\,p$ are contractive $\implies$

we can create recursive protocols using guarded fixpoints

Subprotocols

$$(tag_1, \Phi_1) \sqsubseteq (tag_2, \Phi_2) \triangleq \begin{cases} \forall v.\, \Phi_2\, v \mathrel{-\!\!*} \Phi_1\, v & \text{if } tag_1 = tag_2 = \mathsf{Send} \\ \forall v.\, \Phi_1\, v \mathrel{-\!\!*} \Phi_2\, v & \text{if } tag_1 = tag_2 = \mathsf{Recv} \\ \mathsf{False} & \text{if } tag_1 \neq tag_2 \end{cases}$$

$$c \rightarrowtail p \triangleq \exists q.\, \triangleright(q \sqsubseteq p) * c \xrightarrow{\mathsf{base}} q$$

Subprotocols

$$(tag_1, \Phi_1) \sqsubseteq (tag_2, \Phi_2) \triangleq \begin{cases} \forall v.\, \Phi_2\ v \mathrel{-\!\!*} \Phi_1\ v & \text{if } tag_1 = tag_2 = \text{Send} \\ \forall v.\, \Phi_1\ v \mathrel{-\!\!*} \Phi_2\ v & \text{if } tag_1 = tag_2 = \text{Recv} \\ \text{False} & \text{if } tag_1 \neq tag_2 \end{cases}$$

$$c \rightarrowtail p \triangleq \exists q.\, \triangleright(q \sqsubseteq p) * c \xrightarrow{\text{base}} q$$

Actris subprotocol rules already hold!

$$\frac{\forall x_1.\, \Phi_1\ x_1 \mathrel{-\!\!*} \exists x_2.\, (v_1\ x_1 = v_2\ x_2) * \Phi_2\ x_2 * \triangleright(p_1\ x_1 \sqsubseteq p_2\ x_2)}{\mathbf{?}x_1\ \langle v_1 \rangle \{\Phi_1\}.\, p_1 \sqsubseteq \mathbf{?}x_2\ \langle v_2 \rangle \{\Phi_2\}.\, p_2}$$

$$\frac{\forall x_2.\, \Phi_2\ x_2 \mathrel{-\!\!*} \exists x_1.\, (v_2\ x_2 = v_1\ x_1) * \Phi_1\ x_1 * \triangleright(p_1\ x_1 \sqsubseteq p_2\ x_2)}{\mathbf{!}x_1\ \langle v_1 \rangle \{\Phi_1\}.\, p_1 \sqsubseteq \mathbf{!}x_2\ \langle v_2 \rangle \{\Phi_2\}.\, p_2}$$

(in Actris, these are part of the definition)

Channel closing: three variants

Traditional:

- Separate close/wait: **end**$^!$, **end**$^?$
- Symmetric: **end**

Channel closing: three variants

Traditional:

- Separate close/wait: $\textbf{end}^{!}, \textbf{end}^{?}$
- Symmetric: **end**

Observation: `close` is always forced
Consider $\textbf{!}(b : \textbf{bool})\,\langle b\rangle\{P\}.\,\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ \texttt{end}$
If we send `false`, we **must** then `close`

Channel closing: three variants

Traditional:

- Separate close/wait: $\textbf{end}^{!}, \textbf{end}^{?}$
- Symmetric: **end**

Observation: `close` is always forced
Consider $!\,(b : \textbf{bool})\,\langle b \rangle \{P\}.\ \texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ \texttt{end}$
If we send `false`, we **must** then `close`

Inelegance: allocates a useless `ref`, program contains extraneous close, and does a useless synchronisation.

Channel closing: three variants

Traditional:

- Separate close/wait: $\textbf{end}^{!}, \textbf{end}^{?}$
- Symmetric: **end**

Observation: `close` is always forced
Consider $\mathbf{!}\,(b : \textbf{bool})\,\langle b\rangle\{P\}.\,\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else end}$
If we send `false`, we **must** then `close`

Inelegance: allocates a useless `ref`, program contains extraneous close, and does a useless synchronisation.

Idea: integrated **`send_close`**

- To close, just don't send continuation channel with message
- Other side does ordinary receive, which deallocates
- Most elegant solution (in my opinion)

Comparison with Actris

Actris: channel is a pair of lock-protected buffers
MiniActris: from load & store to channels in 3 simple layers

Actris: relies on garbage collector
MiniActris: uses `free` for receive and channel channel closing

Comparison with Actris

Actris: channel is a pair of lock-protected buffers
MiniActris: from load & store to channels in 3 simple layers

Actris: relies on garbage collector
MiniActris: uses `free` for receive and channel channel closing

Actris: Prot defined using recursive domain equation
MiniActris: non-recursive $\mathsf{Prot} \triangleq \{\mathsf{Send}, \mathsf{Recv}\} \times (\mathsf{Val} \to \mathsf{iProp})$

Actris: higher-order ghost state
MiniActris: only tokens $\mathsf{tok}\,\gamma \triangleq \mathsf{own}\,\gamma\ (\mathsf{Excl}\ ())$

Actris: convenience Coq tactics and notations
MiniActris: no convenience

Comparison with Actris

Actris: channel is a pair of lock-protected buffers
MiniActris: from load & store to channels in 3 simple layers

Actris: relies on garbage collector
MiniActris: uses `free` for receive and channel channel closing

Actris: Prot defined using recursive domain equation
MiniActris: non-recursive $\mathsf{Prot} \triangleq \{\mathsf{Send}, \mathsf{Recv}\} \times (\mathsf{Val} \to \mathsf{iProp})$

Actris: higher-order ghost state
MiniActris: only tokens $\mathsf{tok}\,\gamma \triangleq \mathsf{own}\,\gamma\,(\mathsf{Excl}\,())$

Actris: convenience Coq tactics and notations
MiniActris: no convenience

Actris 2.0: subprotocols + *swapping send/recv*
MiniActris: subprotocols, but swapping *unsound*

Actris 2.0: has reusable language agnostic ghost theory
MiniActris: no ghost theory

Conclusion: Iris ♡ Sessions

MiniActris:

- Simple channel implementation
- Simple invariant
- Simple proofs
- Layered design
- Under 1000 LOC

Suitable as an exercise in separation logic courses?

- Single-shot: yes
- Dependent session protocols: within arm's reach