Dependent Session Protocols in Separation Logic from First Principles

A Separation Logic Proof Pearl

Jules Jacobs
Radboud University Nijmegen

Jonas Kastberg Hinrichsen
Aarhus University

Robbert Krebbers
Radboud University Nijmegen

Message Passing Concurrency

Message passing concurrency:

- Well-structured approach to writing concurrent programs
- Threads as services and clients
- Used in Go, Scala, C#, and more

Message Passing Concurrency

Message passing concurrency:

- Well-structured approach to writing concurrent programs
- Threads as services and clients
- Used in Go, Scala, C#, and more

Bi-directional session channels:

new_chan ()	Create channel and return two endpoints c1 and c2
c.**send**(v)	Send value v over endpoint c
c.**recv**()	Receive and return next inbound value on endpoint c

Message Passing Concurrency

Message passing concurrency:

- Well-structured approach to writing concurrent programs
- Threads as services and clients
- Used in Go, Scala, C#, and more

Bi-directional session channels:

new_chan ()	Create channel and return two endpoints c1 and c2
c.**send**(v)	Send value v over endpoint c
c.**recv**()	Receive and return next inbound value on endpoint c

Example Program:

$$\begin{array}{l}\mathbf{let}\ (c_1, c_2) = \mathbf{new\_chan}\ ()\ \mathbf{in} \\ \mathbf{fork}\ \{\mathbf{let}\ x = c_2.\mathbf{recv}()\ \mathbf{in}\ c_2.\mathbf{send}(x + 2)\}\ ; \\ c_1.\mathbf{send}(40); \mathbf{let}\ y = c_1.\mathbf{recv}()\ \mathbf{in}\ \mathbf{assert}(y = 42)\end{array}$$

Safety and Functional Correctness

Example Program:

$$\mathbf{let}\ (c_1, c_2) = \mathbf{new\_chan}\ ()\ \mathbf{in}$$
$$\mathbf{fork}\ \{\mathbf{let}\ x = c_2.\mathbf{recv}()\ \mathbf{in}\ c_2.\mathbf{send}(x + 2)\}\ ;$$
$$c_1.\mathbf{send}(40);\ \mathbf{let}\ y = c_1.\mathbf{recv}()\ \mathbf{in}\ \mathbf{assert}(y = 42)$$

Goal: Prove crash-freedom (safety) and verify asserts (functional correctness)

Safety and Functional Correctness

Example Program:

$$\textbf{let}\,(c_1, c_2) = \textbf{new\_chan}\,()\ \textbf{in}$$
$$\textbf{fork}\ \{\textbf{let}\ x = c_2.\textbf{recv}()\ \textbf{in}\ c_2.\textbf{send}(x + 2)\}\,;$$
$$c_1.\textbf{send}(40); \textbf{let}\ y = c_1.\textbf{recv}()\ \textbf{in}\ \textbf{assert}(y = 42)$$

Goal: Prove crash-freedom (safety) and verify asserts (functional correctness)

Safety	Functional Correctness
Type systems	Program logics

Safety and Functional Correctness

Example Program:

$$\texttt{let}\,(c_1, c_2) = \texttt{new\_chan}\,()\ \texttt{in}$$
$$\texttt{fork}\ \{\texttt{let}\,x = c_2.\texttt{recv}()\ \texttt{in}\ c_2.\texttt{send}(x+2)\}\,;$$
$$c_1.\texttt{send}(40);\ \texttt{let}\,y = c_1.\texttt{recv}()\ \texttt{in}\ \texttt{assert}(y = 42)$$

Goal: Prove crash-freedom (safety) and verify asserts (functional correctness)

Safety	Functional Correctness
Type systems	Program logics
Automatic checking	Manual proofs

Safety and Functional Correctness

Example Program:

$$\texttt{let}\,(c_1, c_2) = \texttt{new\_chan}\,()\,\texttt{in}$$
$$\texttt{fork}\ \{\texttt{let}\,x = c_2.\texttt{recv}()\,\texttt{in}\,c_2.\texttt{send}(x+2)\}\,;$$
$$c_1.\texttt{send}(40);\,\texttt{let}\,y = c_1.\texttt{recv}()\,\texttt{in}\ \texttt{assert}(y = 42)$$

Goal: Prove crash-freedom (safety) and verify asserts (functional correctness)

Safety	Functional Correctness
Type systems	Program logics
Automatic checking	Manual proofs
Session types	(Actris) dependent session protocols

Example Program:

$$\texttt{let}\,(c_1, c_2) = \texttt{new\_chan}\,()\,\texttt{in}$$
$$\texttt{fork}\,\{\texttt{let}\,x = c_2.\texttt{recv}()\,\texttt{in}\,c_2.\texttt{send}(x+2)\}\,;$$
$$c_1.\texttt{send}(40);\,\texttt{let}\,y = c_1.\texttt{recv}()\,\texttt{in}\,\texttt{assert}(y = 42)$$

Goal: Prove crash-freedom (safety) and verify asserts (functional correctness)

Safety	**Functional Correctness**
Type systems	Program logics
Automatic checking	Manual proofs
Session types	(Actris) dependent session protocols
$!\mathbb{Z}.\,?\mathbb{Z}.\,\textbf{end}$	$!\langle 40\rangle.\,?\langle 42\rangle.\,\textbf{end}$

! is send, **?** is receive

Safety and Functional Correctness

Example Program:

$$\textbf{let}\,(c_1, c_2) = \textbf{new\_chan}\,()\,\textbf{in}$$
$$\textbf{fork}\,\{\textbf{let}\,x = c_2.\textbf{recv}()\,\textbf{in}\,c_2.\textbf{send}(x + 2)\}\,;$$
$$c_1.\textbf{send}(40);\,\textbf{let}\,y = c_1.\textbf{recv}()\,\textbf{in}\,\textbf{assert}(y = 42)$$

Goal: Prove crash-freedom (safety) and verify asserts (functional correctness)

Safety	Functional Correctness
Type systems	Program logics
Automatic checking	Manual proofs
Session types	(Actris) dependent session protocols
$!\mathbb{Z}.\,?\mathbb{Z}.\,\textbf{end}$	$!\,\langle 40\rangle.\,?\langle 42\rangle.\,\textbf{end}$
Minimalist versions exists (Kobayashi et al., Dardha et al.)	Actris employs heavy machinery *Minimalist version is the **goal** of this work*

! is send, **?** is receive

Example Program:

$$\mathbf{let}\,(c_1, c_2) = \mathbf{new\_chan}\,()\ \mathbf{in}$$
$$\mathbf{fork}\ \{\mathbf{let}\,x = c_2.\mathbf{recv}()\ \mathbf{in}\ c_2.\mathbf{send}(x+2)\}\,;$$
$$c_1.\mathbf{send}(40);\ \mathbf{let}\,y = c_1.\mathbf{recv}()\ \mathbf{in}\ \mathbf{assert}(y = 42)$$

Actris dependent session protocols:

$$c_1 \rightarrowtail\ !\langle 40\rangle.\ ?\langle 42\rangle.\ \mathbf{end}$$
$$c_2 \rightarrowtail\ ?\langle 40\rangle.\ !\langle 42\rangle.\ \mathbf{end}$$

Actris Primer

Example Program:

$$
\begin{array}{l}
\textbf{let}\,(c_1, c_2) = \textbf{new\_chan}\,()\,\textbf{in} \\
\textbf{fork}\,\{\textbf{let}\,x = c_2.\textbf{recv}()\,\textbf{in}\,c_2.\textbf{send}(x+2)\}\,; \\
c_1.\textbf{send}(40);\,\textbf{let}\,y = c_1.\textbf{recv}()\,\textbf{in}\,\textbf{assert}(y = 42)
\end{array}
$$

Actris dependent session protocols:

$$
\begin{array}{l}
c_1 \rightarrowtail\, !\,(x : \mathbb{Z})\,\langle x \rangle.\,?\langle x+2 \rangle.\,\textbf{end} \\
c_2 \rightarrowtail\, ?(x : \mathbb{Z})\,\langle x \rangle.\,!\,\langle x+2 \rangle.\,\textbf{end}
\end{array}
$$

Actris Primer

Example Program:

$$\begin{array}{l}\mathbf{let}\,(c_1, c_2) = \mathbf{new\_chan}\,()\,\mathbf{in} \\ \mathbf{fork}\ \{\mathbf{let}\,\ell = c_2.\mathbf{recv}()\,\mathbf{in}\,\ell \leftarrow (!\,\ell + 2); c_2.\mathbf{send}(())\}\,; \\ \mathbf{let}\,\ell = \mathbf{ref}\,40\,\mathbf{in}\,c_1.\mathbf{send}(\ell); c_1.\mathbf{recv}(); \mathbf{assert}(!\,\ell = 42)\end{array}$$

Actris dependent session protocols:

$$c_1 \rightarrowtail ?$$
$$c_2 \rightarrowtail ?$$

Actris Primer

Example Program:

$$\texttt{let}\,(c_1, c_2) = \texttt{new\_chan}\,()\,\texttt{in}$$
$$\texttt{fork}\,\{\texttt{let}\,\ell = c_2.\texttt{recv}()\,\texttt{in}\,\ell \leftarrow (!\,\ell + 2);\, c_2.\texttt{send}(())\}\,;$$
$$\texttt{let}\,\ell = \texttt{ref}\,40\,\texttt{in}\,c_1.\texttt{send}(\ell);\, c_1.\texttt{recv}();\,\texttt{assert}(!\,\ell = 42)$$

Actris dependent session protocols:

$$c_1 \rightarrowtail \mathbf{!}\,(\ell : \mathsf{Loc}, x : \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\, \mathbf{?}\langle () \rangle \{\ell \mapsto (x+2)\}.\,\mathbf{end}$$
$$c_2 \rightarrowtail \mathbf{?}(\ell : \mathsf{Loc}, x : \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\, \mathbf{!}\,\langle () \rangle \{\ell \mapsto (x+2)\}.\,\mathbf{end}$$

Actris Primer

Example Program:

$$\begin{array}{l}\mathtt{let}\,(c_1, c_2) = \mathtt{new\_chan}\,()\,\mathtt{in} \\ \mathtt{fork}\ \{\mathtt{let}\,\ell = c_2.\mathtt{recv}()\,\mathtt{in}\,\ell \leftarrow (!\,\ell + 2); c_2.\mathtt{send}(())\}\,; \\ \mathtt{let}\,\ell = \mathtt{ref}\,40\,\mathtt{in}\,c_1.\mathtt{send}(\ell); c_1.\mathtt{recv}(); \mathtt{assert}(!\,\ell = 42)\end{array}$$

Actris dependent session protocols:

$$\begin{array}{l}c_1 \rightarrowtail\, !\,(\ell : \mathsf{Loc}, x : \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\, ?\langle () \rangle \{\ell \mapsto (x + 2)\}.\,\mathbf{end} \\ c_2 \rightarrowtail\, ?(\ell : \mathsf{Loc}, x : \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\, !\,\langle () \rangle \{\ell \mapsto (x + 2)\}.\,\mathbf{end}\end{array}$$

Actris has many more features:

- Built on top of the Iris higher-order concurrent separation logic framework
  - Allows reasoning about mutable references, locks, and more
- Advanced message passing features
  - Channels as messages, recursive protocols, subprotocols (cf. subtyping)
- Fully mechanised on top of Iris in Coq

Motivation

Observation: Actris is founded upon heavy machinery

- Implementation via custom bi-directional buffers
- Protocols via custom step-indexed recursive domain equation
- Specifications and proofs via custom higher-order ghost state

Motivation

Observation: Actris is founded upon heavy machinery

- Implementation via custom bi-directional buffers
- Protocols via custom step-indexed recursive domain equation
- Specifications and proofs via custom higher-order ghost state

Question: How far can we get with a simpler approach?

Motivation

Observation: Actris is founded upon heavy machinery

- Implementation via custom bi-directional buffers
- Protocols via custom step-indexed recursive domain equation
- Specifications and proofs via custom higher-order ghost state

Question: How far can we get with a simpler approach?

Start from first principles:

- Mutable references ***instead of*** bi-directional buffers
- Higher-order invariants ***instead of*** custom recursive domain equation
- First-order ghost state ***instead of*** higher-order ghost state

Motivation

Observation: Actris is founded upon heavy machinery

- Implementation via custom bi-directional buffers
- Protocols via custom step-indexed recursive domain equation
- Specifications and proofs via custom higher-order ghost state

Question: How far can we get with a simpler approach?

Start from first principles:

- Mutable references ***instead of*** bi-directional buffers
- Higher-order invariants ***instead of*** custom recursive domain equation
- First-order ghost state ***instead of*** higher-order ghost state

All of these features are readily available in Iris!

MiniActris: a Proof Pearl Version of Actris

Key ideas:

1. Build one-shot channels on mutable references
   - With higher-order one-shot protocols via Iris's higher-order invariants
2. Build session channels on one-shot channels (Kobayashi et al., Dardha et al.)
   - With dependent session protocols via nested one-shot protocols
3. Mechanise solution on top of the Iris mechanisation in Coq

MiniActris: a Proof Pearl Version of Actris

Key ideas:

1. Build one-shot channels on mutable references
   - With higher-order one-shot protocols via Iris's higher-order invariants
2. Build session channels on one-shot channels (Kobayashi et al., Dardha et al.)
   - With dependent session protocols via nested one-shot protocols
3. Mechanise solution on top of the Iris mechanisation in Coq

Contributions:

1. A three layered approach to the implementation and specification of channels
   - One-shot channels → functional session channels → imperative session channels
2. Recovering Actris-style specifications for imperative session channels
   - Without custom recursive domain equations or higher-order ghost state
3. A minimalistic mechanisation in **less than 1000 lines** of Coq & Iris code

Outline of Presentation

In the rest of this talk we will cover:

- Layer 1: One-shot channels
- Layer 2: Functional session channels
- Layer 3: Imperative session channels
- Additional features
- Concluding remarks

Layer 1: One-Shot Channels

Layer 1: One-Shot Channels (Implementation)

One-shot channel primitives:

$$\begin{aligned}
\texttt{new1}\,() &\triangleq \texttt{ref None} \\
\texttt{send1}\,c\,v &\triangleq c \leftarrow \texttt{Some}\,v \\
\texttt{recv1}\,c &\triangleq \texttt{match}\ !\,c\ \texttt{with} \\
&\qquad \mid \texttt{None} \Rightarrow \texttt{recv1}\,c \\
&\qquad \mid \texttt{Some}\,v \Rightarrow \texttt{free}\,c;\ v \\
&\qquad \texttt{end}
\end{aligned}$$

Example program:

$$\begin{aligned}
&\texttt{let}\ c = \texttt{new1}\,()\ \texttt{in} \\
&\texttt{fork}\ \{\texttt{let}\ \ell = \texttt{ref}\ 42\ \texttt{in}\ \texttt{send1}\,c\,\ell\}\,; \\
&\texttt{let}\ \ell = \texttt{recv1}\,c\ \texttt{in}\ \texttt{assert}(!\,\ell = 42)
\end{aligned}$$

Layer 1: One-Shot Channels (Specifications)

Protocols and channel permissions:

Protocols: $p ::= (\mathsf{Send}, \Phi) \mid (\mathsf{Recv}, \Phi) \quad \text{where} \quad \Phi : \mathsf{Val} \to \mathsf{Prop}$

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

Permission: $c \rightarrowtail p$

Layer 1: One-Shot Channels (Specifications)

Protocols and channel permissions:

Protocols: $p ::= (\mathsf{Send}, \Phi) \mid (\mathsf{Recv}, \Phi)$ where $\Phi : \mathsf{Val} \to \mathsf{Prop}$

Duality: $\overline{(\mathsf{Send}, \Phi)} \triangleq (\mathsf{Recv}, \Phi)$ $\quad \overline{(\mathsf{Recv}, \Phi)} \triangleq (\mathsf{Send}, \Phi)$

Permission: $c \rightarrowtail p$

(Hoare triple) specifications:

$$\{\mathsf{True}\}\ \mathbf{new1}\,()\ \{c.\, c \rightarrowtail p * c \rightarrowtail \overline{p}\}$$

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

$$\{c \rightarrowtail (\mathsf{Recv}, \Phi)\}\ \mathbf{recv1}\, c\ \{v.\, \Phi\, v\}$$

Example program:

$$\begin{array}{l}\texttt{let}\ c = \texttt{new1}\,()\ \texttt{in} \\ \texttt{fork}\ \{\texttt{let}\ \ell = \texttt{ref}\,42\ \texttt{in}\ \texttt{send1}\,c\,\ell\}\,; \\ \texttt{let}\ \ell = \texttt{recv1}\,c\ \texttt{in}\ \texttt{assert}(!\,\ell = 42)\end{array}$$

Protocol:

$$\begin{array}{l}\Phi\,v \triangleq v \mapsto 42 \\ c \rightarrowtail (\mathsf{Send}, \Phi) \\ c \rightarrowtail (\mathsf{Recv}, \Phi)\end{array}$$

Specifications:

$$\{\mathsf{True}\}\ \texttt{new1}\,()\ \{c.\, c \rightarrowtail p * c \rightarrowtail \overline{p}\}$$

$$\{c \rightarrowtail (\mathsf{Send}, \Phi) * \Phi\,v\}\ \texttt{send1}\,c\,v\ \{\mathsf{True}\}$$

$$\{c \rightarrowtail (\mathsf{Recv}, \Phi)\}\ \texttt{recv1}\,c\ \{v.\, \Phi\,v\}$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.

$$c \rightarrowtail (\mathit{tag}, \Phi) \triangleq \ldots$$

One-shot specifications proven sound with standard Iris methodology.

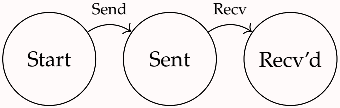
1. Model channel as a state transition system

$$c \rightarrowtail (tag, \Phi) \triangleq \ldots$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.

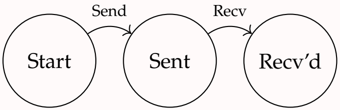
1. Model channel as a state transition system



$$c \rightarrowtail (tag, \Phi) \triangleq \ldots$$

One-shot specifications proven sound with standard Iris methodology.

1. Model channel as a state transition system
2. Capture each state as a disjunct of an invariant

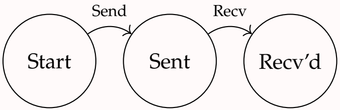


$$c \rightarrowtail (tag, \Phi) \triangleq \ldots$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.

1. Model channel as a state transition system
2. Capture each state as a disjunct of an invariant



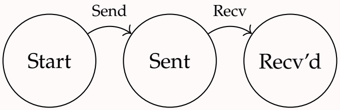
$$\mathsf{chan\_inv} \triangleq (\underbrace{\qquad\qquad}_{\text{(1) initial state}}) \lor (\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\qquad\qquad\qquad}_{\text{(3) final state}})$$

$$c \rightarrowtail (tag, \Phi) \triangleq \ldots$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.

1. Model channel as a state transition system
2. Capture each state as a disjunct of an invariant
3. Determine resource ownership of each state



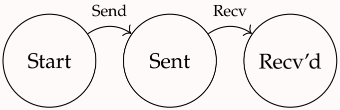
$$\text{chan\_inv} \triangleq (\underbrace{\qquad\qquad}_{\text{(1) initial state}}) \lor (\underbrace{\qquad\qquad\qquad\qquad\qquad}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

$$c \rightarrowtail (tag, \Phi) \triangleq \ldots$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.

1. Model channel as a state transition system
2. Capture each state as a disjunct of an invariant
3. Determine resource ownership of each state



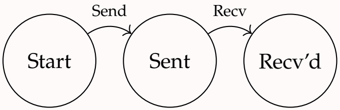
$$\mathsf{chan\_inv}\ c \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v \qquad\qquad}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

$$c \rightarrowtail (tag, \Phi) \triangleq \ldots$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.

1. Model channel as a state transition system
2. Capture each state as a disjunct of an invariant
3. Determine resource ownership of each state



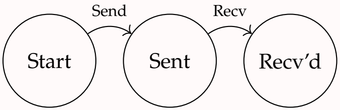
$$\mathsf{chan\_inv} \quad c\,\Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\, v \qquad}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

$$c \rightarrowtail (tag, \Phi) \triangleq \ldots$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.

1. Model channel as a state transition system
2. Capture each state as a disjunct of an invariant
3. Determine resource ownership of each state
4. Encode send/recv transition as transferring a token to the invariant



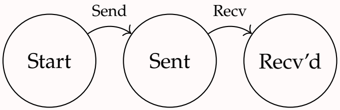
$$\text{chan\_inv} \quad c\,\Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \texttt{Some}\,v * \Phi\,v}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\qquad}_{\text{(3) final state}})$$

$$c \rightarrowtail (tag, \Phi) \triangleq \ldots$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.

1. Model channel as a state transition system
2. Capture each state as a disjunct of an invariant
3. Determine resource ownership of each state
4. Encode send/recv transition as transferring a token to the invariant



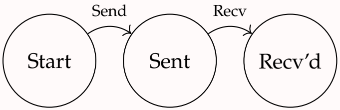
$$\mathsf{chan\_inv}\ \gamma_s \quad c\ \Phi \triangleq (\underbrace{c \mapsto \mathtt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \mathtt{Some}\, v * \Phi\, v * \mathsf{tok}\ \gamma_s}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

$$c \rightarrowtail (\mathit{tag}, \Phi) \triangleq \ldots$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.

1. Model channel as a state transition system
2. Capture each state as a disjunct of an invariant
3. Determine resource ownership of each state
4. Encode send/recv transition as transferring a token to the invariant



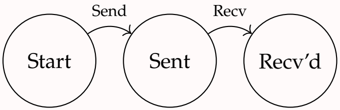
$$\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\, v * \mathsf{tok}\ \gamma_s}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\mathsf{tok}\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

$$c \rightarrowtail (tag, \Phi) \triangleq \ldots$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.

1. Model channel as a state transition system
2. Capture each state as a disjunct of an invariant
3. Determine resource ownership of each state
4. Encode send/recv transition as transferring a token to the invariant
5. Give sender/receiver access to the invariant and their respective token

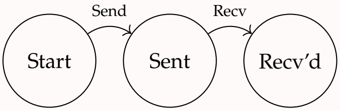


$$\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi \triangleq (\underbrace{c \mapsto \mathtt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \mathtt{Some}\, v * \Phi\, v * \mathsf{tok}\ \gamma_s}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\mathsf{tok}\ \gamma_s * \mathsf{tok}\ \gamma_r}_{\text{(3) final state}})$$

$$c \rightarrowtail (tag, \Phi) \triangleq \ldots$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.



1. Model channel as a state transition system
2. Capture each state as a disjunct of an invariant
3. Determine resource ownership of each state
4. Encode send/recv transition as transferring a token to the invariant
5. Give sender/receiver access to the invariant and their respective token

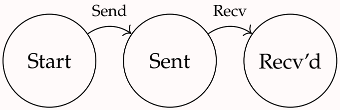
$$\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi \triangleq (\underbrace{c \mapsto \mathbf{None}}_{(1)\text{ initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \mathbf{Some}\ v * \Phi\ v * \mathsf{tok}\ \gamma_s}_{(2)\text{ message sent, but not yet received}}) \vee (\underbrace{\mathsf{tok}\ \gamma_s * \mathsf{tok}\ \gamma_r}_{(3)\text{ final state}})$$

$$c \rightarrowtail (tag, \Phi) \triangleq \exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi}\ \ldots$$

Layer 1: One-Shot Channels (Proof of Specifications)

One-shot specifications proven sound with standard Iris methodology.

1. Model channel as a state transition system
2. Capture each state as a disjunct of an invariant
3. Determine resource ownership of each state
4. Encode send/recv transition as transferring a token to the invariant
5. Give sender/receiver access to the invariant and their respective token



$$\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi \triangleq (\underbrace{c \mapsto \mathtt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \mathtt{Some}\, v * \Phi\ v * \mathsf{tok}\ \gamma_s}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\mathsf{tok}\ \gamma_s * \mathsf{tok}\ \gamma_r}_{\text{(3) final state}})$$

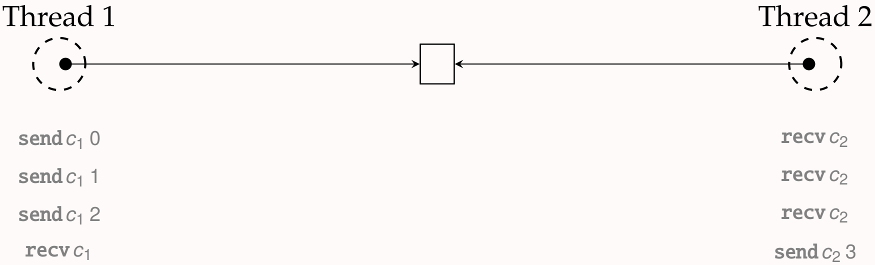
$$c \rightarrowtail (tag, \Phi) \triangleq \exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi} * \triangleright \begin{cases} \mathsf{tok}\ \gamma_s & \text{if } tag = \mathsf{Send} \\ \mathsf{tok}\ \gamma_r & \text{if } tag = \mathsf{Recv} \end{cases}$$

Layer 2: Functional Session Channels

Functional session channel primitives *(Kobayashi et al., Dardha et al.)*:

$$\texttt{new}\,() \triangleq \texttt{new1}\,()$$
$$\texttt{send}\,c\,v \triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\ c'$$
$$\texttt{recv}\,c \triangleq \texttt{recv1}\,c$$

Emerging polarised bi-directional linked list:

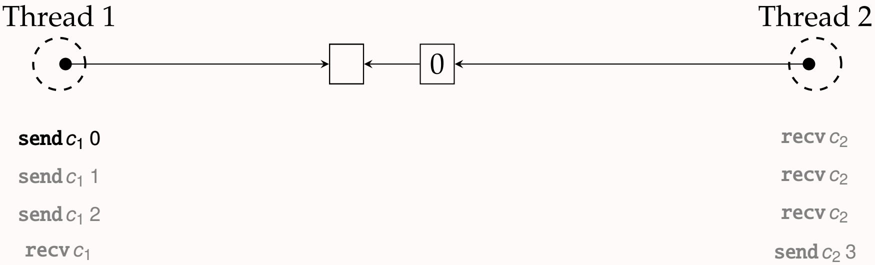


Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives *(Kobayashi et al., Dardha et al.)*:

$$\texttt{new}\,() \triangleq \texttt{new1}\,()$$
$$\texttt{send}\,c\,v \triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\ c'$$
$$\texttt{recv}\,c \triangleq \texttt{recv1}\,c$$

Emerging polarised bi-directional linked list:

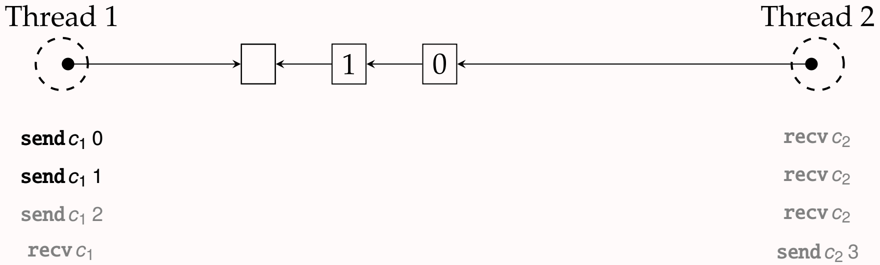


Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives *(Kobayashi et al., Dardha et al.)*:

$$\mathbf{new}\,() \triangleq \mathbf{new1}\,()$$
$$\mathbf{send}\,c\,v \triangleq \mathbf{let}\,c' = \mathbf{new1}\,()\ \mathbf{in}\ \mathbf{send1}\,c\,(v, c');\ c'$$
$$\mathbf{recv}\,c \triangleq \mathbf{recv1}\,c$$

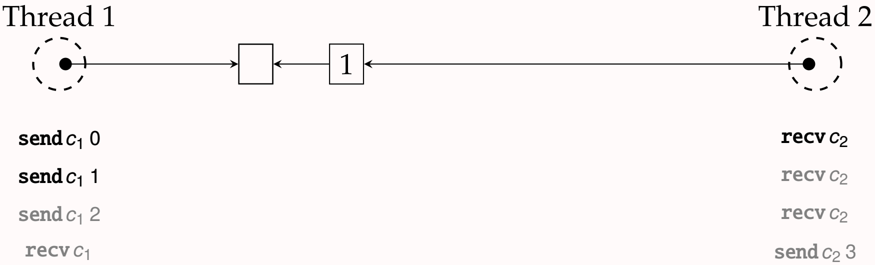
Emerging polarised bi-directional linked list:



Functional session channel primitives *(Kobayashi et al., Dardha et al.)*:

$$\mathtt{new}\,() \triangleq \mathtt{new1}\,()$$
$$\mathtt{send}\,c\,v \triangleq \mathtt{let}\,c' = \mathtt{new1}\,()\,\mathtt{in}\,\mathtt{send1}\,c\,(v, c');\ c'$$
$$\mathtt{recv}\,c \triangleq \mathtt{recv1}\,c$$

Emerging polarised bi-directional linked list:



Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives *(Kobayashi et al., Dardha et al.)***:**

$$\texttt{new}\,() \triangleq \texttt{new1}\,()$$
$$\texttt{send}\,c\,v \triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\ c'$$
$$\texttt{recv}\,c \triangleq \texttt{recv1}\,c$$

Emerging polarised bi-directional linked list:

Thread 1	Thread 2
$\texttt{send}\,c_1\,0$	$\texttt{recv}\,c_2$
$\texttt{send}\,c_1\,1$	$\texttt{recv}\,c_2$
$\texttt{send}\,c_1\,2$	$\texttt{recv}\,c_2$
$\texttt{recv}\,c_1$	$\texttt{send}\,c_2\,3$

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives *(Kobayashi et al., Dardha et al.)*:

$$\mathtt{new}\,() \triangleq \mathtt{new1}\,()$$
$$\mathtt{send}\,c\,v \triangleq \mathtt{let}\,c' = \mathtt{new1}\,()\,\mathtt{in}\,\mathtt{send1}\,c\,(v, c');\ c'$$
$$\mathtt{recv}\,c \triangleq \mathtt{recv1}\,c$$

Emerging polarised bi-directional linked list:

Thread 1	Thread 2
$\mathtt{send}\,c_1\ 0$	$\mathtt{recv}\,c_2$
$\mathtt{send}\,c_1\ 1$	$\mathtt{recv}\,c_2$
$\mathtt{send}\,c_1\ 2$	$\mathtt{recv}\,c_2$
$\mathtt{recv}\,c_1$	$\mathtt{send}\,c_2\ 3$

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives *(Kobayashi et al., Dardha et al.)***:**

$$\texttt{new}\,() \triangleq \texttt{new1}\,()$$
$$\texttt{send}\,c\,v \triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\ c'$$
$$\texttt{recv}\,c \triangleq \texttt{recv1}\,c$$

Emerging polarised bi-directional linked list:

$\texttt{send}\,c_1\,0$	$\texttt{recv}\,c_2$
$\texttt{send}\,c_1\,1$	$\texttt{recv}\,c_2$
$\texttt{send}\,c_1\,2$	$\texttt{recv}\,c_2$
$\texttt{recv}\,c_1$	$\texttt{send}\,c_2\,3$

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives *(Kobayashi et al., Dardha et al.)*:

$$\texttt{new}\,() \triangleq \texttt{new1}\,()$$
$$\texttt{send}\,c\,v \triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\; c'$$
$$\texttt{recv}\,c \triangleq \texttt{recv1}\,c$$

Emerging polarised bi-directional linked list:

Thread 1	Thread 2
$\texttt{send}\,c_1\,0$	$\texttt{recv}\,c_2$
$\texttt{send}\,c_1\,1$	$\texttt{recv}\,c_2$
$\texttt{send}\,c_1\,2$	$\texttt{recv}\,c_2$
$\texttt{recv}\,c_1$	$\texttt{send}\,c_2\,3$

Functional session channel primitives *(Kobayashi et al., Dardha et al.)*:

$$\mathtt{new}\,() \triangleq \mathtt{new1}\,()$$
$$\mathtt{send}\,c\,v \triangleq \mathtt{let}\,c' = \mathtt{new1}\,()\;\mathtt{in}\;\mathtt{send1}\,c\,(v, c');\ c'$$
$$\mathtt{recv}\,c \triangleq \mathtt{recv1}\,c$$

Emerging polarised bi-directional linked list:

Thread 1	Thread 2
$\mathtt{send}\,c_1\,0$	$\mathtt{recv}\,c_2$
$\mathtt{send}\,c_1\,1$	$\mathtt{recv}\,c_2$
$\mathtt{send}\,c_1\,2$	$\mathtt{recv}\,c_2$
$\mathtt{recv}\,c_1$	$\mathtt{send}\,c_2\,3$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\begin{aligned}\mathtt{new}\,() &\triangleq \mathtt{new1}\,() \\ \mathtt{send}\,c\,v &\triangleq \mathtt{let}\,c' = \mathtt{new1}\,()\,\mathtt{in}\,\mathtt{send1}\,c\,(v, c');\ c' \\ \mathtt{recv}\,c &\triangleq \mathtt{recv1}\,c\end{aligned}$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\begin{aligned}\texttt{new}\,() &\triangleq \texttt{new1}\,() \\ \texttt{send}\,c\,v &\triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\; c' \\ \texttt{recv}\,c &\triangleq \texttt{recv1}\,c\end{aligned}$$

Simple session protocols:

$$\mathbf{!}\,\langle w \rangle.\, p \triangleq (\mathsf{Send}, \lambda(v, c').\, v = w * c' \rightarrowtail \overline{p})$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\texttt{new}\,() \triangleq \texttt{new1}\,()$$

$$\texttt{send}\,c\,v \triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\; c'$$

$$\texttt{recv}\,c \triangleq \texttt{recv1}\,c$$

Simple session protocols:

$$\mathbf{!}\,\langle w \rangle.\, p \triangleq (\mathsf{Send}, \lambda(v, c').\, v = w * c' \rightarrowtail \overline{p})$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\texttt{new}\,() \triangleq \texttt{new1}\,()$$
$$\texttt{send}\,c\,v \triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\; c'$$
$$\texttt{recv}\,c \triangleq \texttt{recv1}\,c$$

Simple session protocols:

$$!\,\langle w \rangle.\,p \triangleq (\mathsf{Send}, \lambda(v, c').\, v = w * c' \rightarrowtail \overline{p})$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\begin{aligned}\texttt{new}\,() &\triangleq \texttt{new1}\,() \\ \texttt{send}\,c\,v &\triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\; c' \\ \texttt{recv}\,c &\triangleq \texttt{recv1}\,c\end{aligned}$$

Simple session protocols:

$$\mathbf{!}\,\langle w\rangle.\,p \triangleq (\mathsf{Send}, \lambda(v, c').\, v = w * c' \rightarrowtail \overline{p})$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\mathtt{new}\,() \triangleq \mathtt{new1}\,()$$

$$\mathtt{send}\,c\,v \triangleq \mathtt{let}\,c' = \mathtt{new1}\,()\,\mathtt{in}\,\mathtt{send1}\,c\,(v, c');\ c'$$

$$\mathtt{recv}\,c \triangleq \mathtt{recv1}\,c$$

Dependent session protocols:

$$!\,(x : \tau)\,\langle w\rangle\{P\}.\,p \triangleq (\mathsf{Send}, \lambda(v, c').\,\exists(x : \tau).\,v = (w\ x) * P\,x * c' \rightarrowtail \overline{p\,x})$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\mathtt{new}\,() \triangleq \mathtt{new1}\,()$$
$$\mathtt{send}\,c\,v \triangleq \mathtt{let}\,c' = \mathtt{new1}\,()\,\mathtt{in}\,\mathtt{send1}\,c\,(v, c');\; c'$$
$$\mathtt{recv}\,c \triangleq \mathtt{recv1}\,c$$

Dependent session protocols:

$$!\,(x : \tau)\,\langle w \rangle \{P\}.\, p \triangleq (\mathsf{Send}, \lambda(v, c').\, \exists (x : \tau).\, v = (w\ x) * P\ x * c' \rightarrowtail \overline{p\ x})$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\begin{aligned} \texttt{new}\,() &\triangleq \texttt{new1}\,() \\ \texttt{send}\,c\,v &\triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\; c' \\ \texttt{recv}\,c &\triangleq \texttt{recv1}\,c \end{aligned}$$

Dependent session protocols:

$$!\,(x : \tau)\,\langle w \rangle \{P\}.\,p \triangleq (\mathsf{Send}, \lambda(v, c').\, \exists (x : \tau).\, v = (w\;x) * P\;x * c' \rightarrowtail \overline{p\;x})$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\begin{aligned}
\texttt{new}\,() &\triangleq \texttt{new1}\,() \\
\texttt{send}\,c\,v &\triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\; c' \\
\texttt{recv}\,c &\triangleq \texttt{recv1}\,c
\end{aligned}$$

Dependent session protocols:

$$\begin{aligned}
{!}\,(x : \tau)\,\langle w \rangle \{P\}.\, p &\triangleq (\mathsf{Send}, \lambda(v, c').\, \exists (x : \tau).\, v = (w\,x) * P\,x * c' \rightarrowtail \overline{p\,x}) \\
{?}\,(x : \tau)\,\langle w \rangle \{P\}.\, p &\triangleq \overline{{!}\,(x : \tau)\,\langle w \rangle \{P\}.\, \overline{p}}
\end{aligned}$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\begin{aligned}\mathtt{new}\,() &\triangleq \mathtt{new1}\,() \\ \mathtt{send}\,c\,v &\triangleq \mathtt{let}\,c' = \mathtt{new1}\,()\,\mathtt{in}\,\mathtt{send1}\,c\,(v, c');\; c' \\ \mathtt{recv}\,c &\triangleq \mathtt{recv1}\,c\end{aligned}$$

Dependent session protocols:

$$\begin{aligned}\mathbf{!}\,(x : \tau)\,\langle w\rangle\{P\}.\,p &\triangleq (\mathsf{Send}, \lambda(v, c').\,\exists(x:\tau).\,v = (w\,x) * P\,x * c' \rightarrowtail \overline{p\,x}) \\ \mathbf{?}\,(x : \tau)\,\langle w\rangle\{P\}.\,p &\triangleq \overline{\mathbf{!}\,(x : \tau)\,\langle w\rangle\{P\}.\,\overline{p}}\end{aligned}$$

Specifications:

$$\begin{aligned}\{\mathsf{True}\}\;&\mathtt{new}\,()\;\{c.\,c \rightarrowtail p * c \rightarrowtail \overline{p}\} \\ \{c \rightarrowtail (\mathbf{!}\,(x : \tau)\,\langle w\rangle\{P\}.\,p) * P\,t\}\;&\mathtt{send}\,c\,(w\,t)\;\{c'.\,c' \rightarrowtail p\,t\} \\ \{c \rightarrowtail (\mathbf{?}\,(x : \tau)\,\langle w\rangle\{P\}.\,p)\}\;&\mathtt{recv}\,c\;\{(v, c').\,\exists(x:\tau).\,v = (w\,x) * P\,x * c' \rightarrowtail p\,x\}\end{aligned}$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\begin{aligned}
\texttt{new}\,() &\triangleq \texttt{new1}\,() \\
\texttt{send}\,c\,v &\triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\; c' \\
\texttt{recv}\,c &\triangleq \texttt{recv1}\,c
\end{aligned}$$

Dependent session protocols:

$$\begin{aligned}
{!}\,(x:\tau)\,\langle w\rangle\{P\}.\,p &\triangleq (\mathsf{Send}, \lambda(v, c').\, \exists(x:\tau).\, v = (w\;x) * P\;x * c' \rightarrowtail \overline{p\;x}) \\
{?}\,(x:\tau)\,\langle w\rangle\{P\}.\,p &\triangleq \overline{{!}\,(x:\tau)\,\langle w\rangle\{P\}.\,\overline{p}}
\end{aligned}$$

Specifications:

$$\begin{aligned}
\{\mathsf{True}\}\;\; &\texttt{new}\,()\;\; \{c.\, c \rightarrowtail p * c \rightarrowtail \overline{p}\} \\
\{c \rightarrowtail ({!}\,(x:\tau)\,\langle w\rangle\{P\}.\,p) * P\;t\}\;\; &\texttt{send}\,c\,(w\;t)\;\; \{c'.\, c' \rightarrowtail p\;t\} \\
\{c \rightarrowtail ({?}\,(x:\tau)\,\langle w\rangle\{P\}.\,p)\}\;\; &\texttt{recv}\,c\;\; \{(v, c').\, \exists(x:\tau).\, v = (w\;x) * P\;x * c' \rightarrowtail p\;x\}
\end{aligned}$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\begin{aligned}
\mathtt{new}\,() &\triangleq \mathtt{new1}\,() \\
\mathtt{send}\,c\,v &\triangleq \mathtt{let}\,c' = \mathtt{new1}\,()\,\mathtt{in}\,\mathtt{send1}\,c\,(v, c');\; c' \\
\mathtt{recv}\,c &\triangleq \mathtt{recv1}\,c
\end{aligned}$$

Dependent session protocols:

$$\begin{aligned}
{!}\,(x:\tau)\,\langle w\rangle\{P\}.\,p &\triangleq (\mathsf{Send}, \lambda(v, c').\,\exists(x:\tau).\,v = (w\;x) * P\;x * c' \rightarrowtail \overline{p\;x}) \\
{?}\,(x:\tau)\,\langle w\rangle\{P\}.\,p &\triangleq \overline{{!}\,(x:\tau)\,\langle w\rangle\{P\}.\,\overline{p}}
\end{aligned}$$

Specifications:

$$\begin{aligned}
\{\mathsf{True}\}\;\; &\mathtt{new}\,()\;\; \{c.\,c \rightarrowtail p * c \rightarrowtail \overline{p}\} \\
\{c \rightarrowtail ({!}\,(x:\tau)\,\langle w\rangle\{P\}.\,p) * P\;t\}\;\; &\mathtt{send}\,c\,(w\;t)\;\; \{c'.\,c' \rightarrowtail p\;t\} \\
\{c \rightarrowtail ({?}\,(x:\tau)\,\langle w\rangle\{P\}.\,p)\}\;\; &\mathtt{recv}\,c\;\; \{(v, c').\,\exists(x:\tau).\,v = (w\;x) * P\;x * c' \rightarrowtail p\;x\}
\end{aligned}$$

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

$$\texttt{new}\,() \triangleq \texttt{new1}\,()$$
$$\texttt{send}\,c\,v \triangleq \texttt{let}\,c' = \texttt{new1}\,()\,\texttt{in}\,\texttt{send1}\,c\,(v, c');\; c'$$
$$\texttt{recv}\,c \triangleq \texttt{recv1}\,c$$

Dependent session protocols:

$$!\,(x : \tau)\,\langle w \rangle\{P\}.\,p \triangleq (\mathsf{Send}, \lambda(v, c').\,\exists(x : \tau).\,v = (w\;x) * P\;x * c' \rightarrowtail \overline{p\;x})$$
$$?\,(x : \tau)\,\langle w \rangle\{P\}.\,p \equiv (\mathsf{Recv}, \lambda(v, c').\,\exists(x : \tau).\,v = (w\;x) * P\;x * c' \rightarrowtail p\;x)$$

Specifications:

$$\{\mathsf{True}\}\;\texttt{new}\,()\;\{c.\,c \rightarrowtail p * c \rightarrowtail \overline{p}\}$$
$$\{c \rightarrowtail (!\,(x : \tau)\,\langle w \rangle\{P\}.\,p) * P\;t\}\;\texttt{send}\,c\,(w\;t)\;\{c'.\,c' \rightarrowtail p\;t\}$$
$$\{c \rightarrowtail (?\,(x : \tau)\,\langle w \rangle\{P\}.\,p)\}\;\texttt{recv}\,c\;\{(v, c').\,\exists(x : \tau).\,v = (w\;x) * P\;x * c' \rightarrowtail p\;x\}$$

Layer 2: Functional Session Channels (Crux of Session Protocols)

Observation: *Dependent session protocol definitions rely on higher-order invariants*

Layer 2: Functional Session Channels (Crux of Session Protocols)

Observation: *Dependent session protocol definitions rely on higher-order invariants*

Recall the following definitions:

$$c \rightarrowtail p \triangleq \exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ p.2}\ \ldots$$

$$\mathbf{!}\,(x : \tau)\,\langle w \rangle \{P\}.\,p \triangleq (\mathsf{Send}, \lambda(v, c').\ \exists (x : \tau).\ c' \rightarrowtail \overline{p\ x} \ldots)$$

Layer 2: Functional Session Channels (Crux of Session Protocols)

Observation: *Dependent session protocol definitions rely on higher-order invariants*

Recall the following definitions:

$$c \rightarrowtail p \triangleq \exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ p.2}\ \ldots$$

$$!\,(x : \tau)\,\langle w \rangle \{P\}.\,p \triangleq (\mathsf{Send}, \lambda(v, c').\ \exists (x : \tau).\ c' \rightarrowtail \overline{p\ x} \ldots)$$

Unfolding the definitions yield the following nesting:

$c \rightarrowtail\ !\,(x : \tau)\,\langle w \rangle \{P\}.\,p$

Layer 2: Functional Session Channels (Crux of Session Protocols)

Observation: *Dependent session protocol definitions rely on higher-order invariants*

Recall the following definitions:

$$c \rightarrowtail p \triangleq \exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ p.2}\ \ldots$$

$$!\,(x : \tau)\,\langle w \rangle \{P\}.\,p \triangleq (\mathsf{Send}, \lambda(v, c').\ \exists (x : \tau).\ c' \rightarrowtail \overline{p\ x} \ldots)$$

Unfolding the definitions yield the following nesting:

$$c \rightarrowtail\ !\,(x : \tau)\,\langle w \rangle \{P\}.\,p\ \equiv$$
$$\exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ (!\,(x : \tau)\,\langle w \rangle \{P\}.\,p).2}\ \ldots$$

Layer 2: Functional Session Channels (Crux of Session Protocols)

Observation: *Dependent session protocol definitions rely on higher-order invariants*

Recall the following definitions:

$$c \rightarrowtail p \triangleq \exists \gamma_s, \gamma_r. \boxed{\textsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ p.2}\ \ldots$$

$$\mathbf{!}\,(x : \tau)\,\langle w \rangle \{P\}.\, p \triangleq (\textsf{Send}, \lambda(v, c').\, \exists (x : \tau).\, c' \rightarrowtail \overline{p\ x} \ldots)$$

Unfolding the definitions yield the following nesting:

$$c \rightarrowtail \mathbf{!}\,(x : \tau)\,\langle w \rangle \{P\}.\, p \ \equiv$$
$$\exists \gamma_s, \gamma_r. \boxed{\textsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ (\mathbf{!}\,(x : \tau)\,\langle w \rangle \{P\}.\, p).2}\ \ldots\ \equiv$$
$$\exists \gamma_s, \gamma_r. \boxed{\textsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ (\lambda(v, c').\, \exists (x : \tau).\, c' \rightarrowtail \overline{p\ x} \ldots)}\ \ldots$$

Layer 2: Functional Session Channels (Crux of Session Protocols)

Observation: *Dependent session protocol definitions rely on higher-order invariants*

Recall the following definitions:

$$c \rightarrowtail p \triangleq \exists \gamma_s, \gamma_r. \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ p.2}\ \ldots$$

$$!\,(x : \tau)\,\langle w \rangle \{P\}.\, p \triangleq (\mathsf{Send}, \lambda(v, c').\, \exists (x : \tau).\, c' \rightarrowtail \overline{p\ x} \ldots)$$

Unfolding the definitions yield the following nesting:

$$
\begin{aligned}
& c \rightarrowtail !\,(x : \tau)\,\langle w \rangle \{P\}.\, p \ \equiv \\
& \exists \gamma_s, \gamma_r. \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ (!\,(x : \tau)\,\langle w \rangle \{P\}.\, p).2}\ \ldots \ \equiv \\
& \exists \gamma_s, \gamma_r. \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ (\lambda(v, c').\, \exists (x : \tau).\, c' \rightarrowtail \overline{p\ x} \ldots)}\ \ldots \ \equiv \\
& \exists \gamma_s, \gamma_r. \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ (\lambda(v, c').\, \exists (x : \tau).\, \exists \gamma_s, \gamma_r. \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c'\ (\overline{p\ x}).2}\ \ldots)}\ \ldots
\end{aligned}
$$

Layer 2: Functional Session Channels (Crux of Session Protocols)

Observation: *Dependent session protocol definitions rely on higher-order invariants*

Recall the following definitions:

$$c \rightarrowtail p \triangleq \exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ p.2}\ \ldots$$
$$!\,(x : \tau)\,\langle w \rangle \{P\}.\,p \triangleq (\mathsf{Send}, \lambda(v, c').\,\exists (x : \tau).\,c' \rightarrowtail \overline{p\ x} \ldots)$$

Unfolding the definitions yield the following nesting:

$$
\begin{aligned}
& c \rightarrowtail !\,(x : \tau)\,\langle w \rangle \{P\}.\,p \ \equiv \\
& \exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ (!\,(x : \tau)\,\langle w \rangle \{P\}.\,p).2}\ \ldots \ \equiv \\
& \exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ (\lambda(v, c').\,\exists (x : \tau).\,c' \rightarrowtail \overline{p\ x} \ldots)}\ \ldots \ \equiv \\
& \exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ (\lambda(v, c').\,\exists (x : \tau).\,\exists \gamma_s, \gamma_r.\ \boxed{\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c'\ (\overline{p\ x}).2}\ \ldots)}\ \ldots
\end{aligned}
$$

Nested invariants *are readily supported by Iris*

Layer 3: Imperative Channels

Layer 3: Imperative Channels (Motivation and Implementation)

Functional channels are inconvenient:

$$\mathbf{let}\, c = \mathbf{send}\, c\, v\ \mathbf{in}\ \mathbf{recv}\, c$$

We instead want:

$$c.\mathbf{send}(v);\, c.\mathbf{recv}()$$

Layer 3: Imperative Channels (Motivation and Implementation)

Functional channels are inconvenient:

$$\texttt{let}\, c = \texttt{send}\, c\, v\, \texttt{in}\, \texttt{recv}\, c$$

We instead want:

$$c.\texttt{send}(v);\, c.\texttt{recv}()$$

Solution: Imperative channels

$$\begin{aligned} \texttt{new\_chan}\,() &\triangleq \texttt{let}\, c = \texttt{new}\,()\, \texttt{in}\, (\texttt{ref}\, c, \texttt{ref}\, c) \\ c.\texttt{send}(v) &\triangleq c \leftarrow \texttt{send}\, (!\, c)\, v \\ c.\texttt{recv}() &\triangleq \texttt{let}\, (v, c') = \texttt{recv}\, !\, c\, \texttt{in}\, c \leftarrow c';\, v \end{aligned}$$

Layer 3: Imperative Channels (Motivation and Implementation)

Functional channels are inconvenient:

$$\mathbf{let}\, c = \mathbf{send}\, c\, v\, \mathbf{in}\, \mathbf{recv}\, c$$

We instead want:

$$c.\mathbf{send}(v);\, c.\mathbf{recv}()$$

Solution: Imperative channels

$$\mathbf{new\_chan}\,() \triangleq \mathbf{let}\, c = \mathbf{new}\,()\,\mathbf{in}\,(\mathbf{ref}\, c, \mathbf{ref}\, c)$$
$$c.\mathbf{send}(v) \triangleq c \leftarrow \mathbf{send}\,(!\, c)\, v$$
$$c.\mathbf{recv}() \triangleq \mathbf{let}\,(v, c') = \mathbf{recv}\, !\, c\, \mathbf{in}\, c \leftarrow c';\, v$$

With this we can write the program from the introduction:

$$\begin{array}{l} \mathbf{let}\,(c_1, c_2) = \mathbf{new\_chan}\,()\,\mathbf{in} \\ \mathbf{fork}\,\{\mathbf{let}\,\ell = c_2.\mathbf{recv}()\,\mathbf{in}\,\ell \leftarrow (!\,\ell + 2);\, c_2.\mathbf{send}(())\}\,; \\ \mathbf{let}\,\ell = \mathbf{ref}\,40\,\mathbf{in}\,c_1.\mathbf{send}(\ell);\, c_1.\mathbf{recv}();\, \mathbf{assert}(!\,\ell = 42) \end{array}$$

Layer 3: Imperative Channels (Specifications)

Imperative channel endpoint ownership:

$$c \overset{\mathsf{imp}}{\rightarrowtail} p \triangleq \exists (c' : \mathsf{Val}).\, c \mapsto c' * c' \rightarrowtail p$$

Layer 3: Imperative Channels (Specifications)

Imperative channel endpoint ownership:

$$c \xrightarrowtail{\text{imp}} p \triangleq \exists (c' : \mathsf{Val}).\, c \mapsto c' * c' \rightarrowtail p$$

Actris specifications:

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

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

$$\{c \xrightarrowtail{\text{imp}} (\mathbf{?}(x : \tau)\,\langle w \rangle \{P\}.\, p)\}\ c.\mathbf{recv}()\ \{v.\, \exists (x : \tau).\, v = (w\ x) * P\ x * c \xrightarrowtail{\text{imp}} p\ x\}$$

Layer 3: Imperative Channels (Specifications)

Imperative channel endpoint ownership:

$$c \xrightarrowtail{\text{imp}} p \triangleq \exists (c' : \mathsf{Val}).\, c \mapsto c' * c' \rightarrowtail p$$

Actris specifications:

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

$$\{c \xrightarrowtail{\text{imp}} (!\,(x : \tau)\,\langle w \rangle \{P\}.\, p) * P\ t\}\ c.\mathtt{send}(w\ t)\ \{c \xrightarrowtail{\text{imp}} p\ t\}$$

$$\{c \xrightarrowtail{\text{imp}} (?(x : \tau)\,\langle w \rangle \{P\}.\, p)\}\ c.\mathtt{recv}()\ \{v.\, \exists (x : \tau).\, v = (w\ x) * P\ x * c \xrightarrowtail{\text{imp}} p\ x\}$$

Proof of specifications is trivial reasoning about references

Layer 3: Imperative Channels (Proof of Example)

Program from introduction:

$$\mathtt{let}\,(c_1, c_2) = \mathtt{new\_chan}\,()\,\mathtt{in}$$
$$\mathtt{fork}\ \{\mathtt{let}\,\ell = c_2.\mathtt{recv}()\,\mathtt{in}\,\ell \leftarrow (!\,l + 2); c_2.\mathtt{send}(())\}\,;$$
$$\mathtt{let}\,\ell = \mathtt{ref}\,40\,\mathtt{in}\,c_1.\mathtt{send}(\ell); c_1.\mathtt{recv}(); \mathtt{assert}(!\,\ell = 42)$$

Protocols:[1]

$$c_1 \overset{\mathsf{imp}}{\rightarrowtail} \mathbf{!}\,(\ell : \mathsf{Loc}, x : \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\, \mathbf{?}\langle () \rangle \{\ell \mapsto (x+2)\}.\, \mathbf{?end}$$
$$c_2 \overset{\mathsf{imp}}{\rightarrowtail} \mathbf{?}(\ell : \mathsf{Loc}, x : \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\, \mathbf{!}\,\langle () \rangle \{\ell \mapsto (x+2)\}.\, \mathbf{!end}$$

Actris specifications:

$$\{\mathsf{True}\}\ \mathtt{new\_chan}\,()\ \{(c_1, c_2).\, c_1 \overset{\mathsf{imp}}{\rightarrowtail} p * c_2 \overset{\mathsf{imp}}{\rightarrowtail} \overline{p}\}$$
$$\{c \overset{\mathsf{imp}}{\rightarrowtail} (\mathbf{!}\,(x : \tau)\,\langle w \rangle \{P\}.\, p) * P\ t\}\ c.\mathtt{send}(w\ t)\ \{c \overset{\mathsf{imp}}{\rightarrowtail} p\ t\}$$
$$\{c \overset{\mathsf{imp}}{\rightarrowtail} (\mathbf{?}(x : \tau)\,\langle w \rangle \{P\}.\, p)\}\ c.\mathtt{recv}()\ \{v.\, \exists (x : \tau).\, v = (w\ x) * P\ x * c \overset{\mathsf{imp}}{\rightarrowtail} p\ x\}$$

---

[1] $\mathbf{!end} \triangleq (\mathsf{Send}, \lambda v.\, v = ())$ | $\mathbf{?end} \triangleq \overline{\mathbf{!end}}$

Additional Features of MiniActris

Additional Features of MiniActris

Recursive protocols: $\mu p.\, \mathbf{!}\,\langle 40\rangle.\, \mathbf{?}\langle 42\rangle.\, p$

Variance subprotocols: $\mathbf{?}(n : \mathbb{N})\,\langle n\rangle.\, \mathbf{!}\,\langle n+2\rangle.\, p \;\sqsubseteq\; \mathbf{?}(x : \mathbb{Z})\,\langle x\rangle.\, \mathbf{!}\,\langle x+2\rangle.\, p$

Channel deallocation: traditional (symmetric, asymmetric) & new (closing sends)

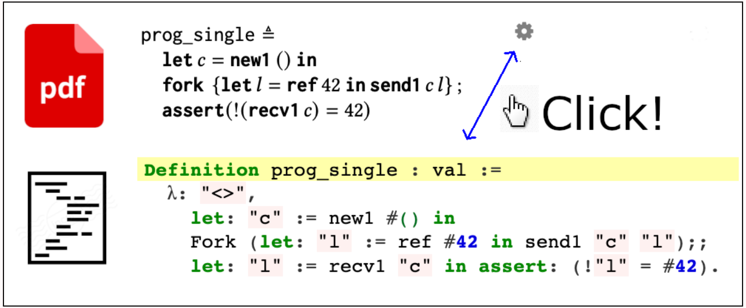
Additional Features of MiniActris

Recursive protocols: $\mu p.\,!\,\langle 40 \rangle.\,?\langle 42 \rangle.\,p$

Variance subprotocols: $?(n : \mathbb{N})\,\langle n \rangle.\,!\,\langle n+2 \rangle.\,p \;\sqsubseteq\; ?(x : \mathbb{Z})\,\langle x \rangle.\,!\,\langle x+2 \rangle.\,p$

Channel deallocation: traditional (symmetric, asymmetric) & new (closing sends)

Everything mechanized in less than 1000 lines of Coq!



Concluding Remarks

Comparison with Actris

MiniActris

This work

(ICFP'23)

- **Asynchronous channels**
- **Dependent session protocols**
- **Iris separation logic**
- **Channels as messages**
- **Recursive protocols**
- **Channel deallocation**
- **Variance subprotocols**

Comparison with Actris

MiniActris
This work
(ICFP'23)

- **Asynchronous channels**
- **Dependent session protocols**
- **Iris separation logic**
- **Channels as messages**
- **Recursive protocols**
- **Channel deallocation**
- **Variance subprotocols**

Actris 1.0
Hinrichsen, Bengtson, Krebbers
(POPL'20)

Comparison with Actris

MiniActris
This work
(ICFP'23)

- **Asynchronous channels**
- **Dependent session protocols**
- **Iris separation logic**
- **Channels as messages**
- **Recursive protocols**
- **Channel deallocation**
- **Variance subprotocols**
- **Asynchronous subprotocols**
- **Actris ghost theory**

Actris 1.0 (Asynchronous channels – Recursive protocols)
Hinrichsen, Bengtson, Krebbers
(POPL'20)

Actris 2.0 (Variance subprotocols – Actris ghost theory)
Hinrichsen, Bengtson, Krebbers
(LMCS'22)

Comparison with Actris

MiniActris

This work

(ICFP'23)

Asynchronous channels

Dependent session protocols

Iris separation logic

Channels as messages

Recursive protocols

Actris 1.0

Hinrichsen, Bengtson, Krebbers

(POPL'20)

Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols

LÉON GONDELMAN, Aarhus University, Denmark
JONAS KASTBERG HINRICHSEN, Aarhus University, Denmark
MÁRIO PEREIRA, NOVA LINCS, NOVA School of Science and Technology, Portugal
AMIN TIMANY, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

n

s

cols

Actris ghost theory

Actris 2.0

Hinrichsen, Bengtson, Krebbers

(LMCS'22)

Conclusion: Sessions ♡ (Iris) Higher-Order Separation Logic

MiniActris: ***a separation logic proof pearl for verified message passing***

- Three layers: one-shot → functional → imperative
- Simple soundness proof with nested invariants
- Abundance of protocol features
- Mechanized in less than 1000 lines of Coq code

Suitable as an exercise in separation logic courses?

- One-shot channels: *suitable*
- Session channels: *within arms reach*

$$
\begin{aligned}
&\mathbf{!}\,\langle\text{“Thank you”}\rangle\{\texttt{MiniActrisKnowledge}\}.\\
&\mu\texttt{rec}.\ \mathbf{?}(q : \texttt{Question})\,\langle q\rangle\{\texttt{AboutMiniActris}\ q\}.\\
&\qquad\quad \mathbf{!}\,(a : \texttt{Answer})\,\langle a\rangle\{\texttt{Insightful}\ q\ a\}.\,\mathit{rec}
\end{aligned}
$$

Backup Slides

Distributed MiniActris?

Conjecture: Not as elegant

- Handshake when creating new one-shot channels is non-trivial at scale
- Might be solved with session context, but then one-shots make less sense

MiniActris Ghost Theory?

Conjecture: Not feasible

- The recursion in MiniActris is tied by the references of the program
- A ghost theory solution would need to explicitly track the linked list
- Quickly ends up with similar workload as current Actris ghost theory