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:

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

Message Passing Concurrency

Message passing:

- 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:

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

Bi-directional session channels:

$\mathbf{new\_chan}\ ()$	Create channel and return two endpoints c1 and c2
$c.\mathbf{send}(v)$	Send value v over endpoint c
$c.\mathbf{recv}()$	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:

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

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

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

Safety and Functional Correctness

Example Program:

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

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:

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

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

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
$!\mathbb{Z}.\,?\mathbb{Z}.\,\mathbf{end}$	$!\,\langle 40\rangle.\,?\langle 42\rangle.\,\mathbf{end}$

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
$!\mathbb{Z}.\,?\mathbb{Z}.\,\texttt{end}$	$!\,\langle 40\rangle.\,?\langle 42\rangle.\,\texttt{end}$
Minimalist versions exists (Dhardha et al., Kobayashi et al.)	Actris employs heavy machinery *Minimalist version is the **goal** of this work*

Example Program:

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

Actris dependent session protocols:

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

Actris Primer

Example Program:

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

Actris dependent session protocols:

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

Example Program:

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

Actris dependent session protocols:

$$\begin{aligned}
c_1 &\rightarrowtail ? \\
c_2 &\rightarrowtail ?
\end{aligned}$$

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:

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

Actris dependent session protocols:

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

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 over channels, recursive protocols, subprotocols (cf. subtyping)
- Fully mechanised on top of Iris in Coq

Motivation

Observation: Actris is founded upon heavy machinery

- Custom bi-directional buffer implementation of session channels
- Custom step-indexed recursive domain equation to obtain protocols
- Custom higher-order ghost state

Motivation

Observation: Actris is founded upon heavy machinery

- Custom bi-directional buffer implementation of session channels
- Custom step-indexed recursive domain equation to obtain protocols
- Custom higher-order ghost state

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

Motivation

Observation: Actris is founded upon heavy machinery

- Custom bi-directional buffer implementation of session channels
- Custom step-indexed recursive domain equation to obtain protocols
- 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

- Custom bi-directional buffer implementation of session channels
- Custom step-indexed recursive domain equation to obtain protocols
- 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 support in Iris!

MiniActris: a Proof Pearl Version of Actris

Key ideas:

1. Build one-shot channels on mutable references,
   and higher-order one-shot protocols on Iris's higher-order invariants
2. Build session channels on one-shot channels (Kobayashi et al., Dharda et al.),
   and session protocols on 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,
   and higher-order one-shot protocols on Iris's higher-order invariants
2. Build session channels on one-shot channels (Kobayashi et al., Dharda et al.),
   and session protocols on 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)

Channel primitives:

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

Example program:

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

Layer 1: One-Shot Channels (Specifications)

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

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

Points-to: $c \rightarrowtail p$

Layer 1: One-Shot Channels (Specifications)

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

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

Points-to: $c \rightarrowtail p$

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:

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

Protocol:

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

Specifications:

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

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

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

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.

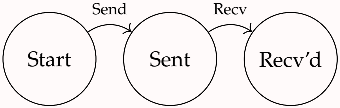
1. Model channel as a state transition system

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

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

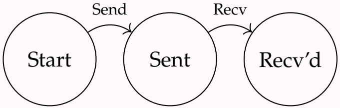


$$c \rightarrowtail (tag, \Phi) \triangleq \ldots$$

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

One-shot specifications proven sound with standard Iris methodology.

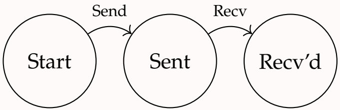
1. Model channel as a state transition system
2. Use invariant with disjunct for each state



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

One-shot specifications proven sound with standard Iris methodology.

1. Model channel as a state transition system
2. Use invariant with disjunct for each state



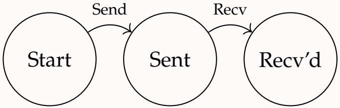
$$\mathsf{chan\_inv} \triangleq (\underbrace{\qquad\qquad}_{\text{(1) initial state}}) \lor (\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

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

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. Use invariant with disjunct for each state
3. Determine resource ownership of each state



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

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

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. Use invariant with disjunct for each state
3. Determine resource ownership of each state

$$\text{Start} \xrightarrow{\text{Send}} \text{Sent} \xrightarrow{\text{Recv}} \text{Recv'd}$$

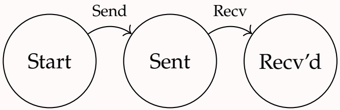
$$\mathsf{chan\_inv} \quad c\ \Phi \triangleq (\underbrace{c \mapsto \mathtt{None}}_{\text{(1) initial state}}) \lor (\underbrace{\exists v.\, c \mapsto \mathtt{Some}\, v * \Phi\ v}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

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

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. Use invariant with disjunct for each state
3. Determine resource ownership of each state
4. Encode send/recv transition as transferring a token to the invariant

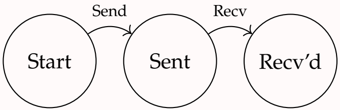


$$\mathsf{chan\_inv} \quad c\,\Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \lor (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\, v \qquad}_{\text{(2) message sent, but not yet received}}) \lor (\underbrace{\qquad\qquad}_{\text{(3) final state}})$$

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

One-shot specifications proven sound with standard Iris methodology.

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



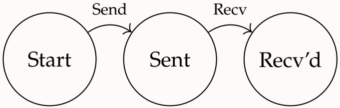
$$\mathsf{chan\_inv}\ \gamma_s \quad c\ \Phi \triangleq (\underbrace{c \mapsto \texttt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \texttt{Some}\, v * \Phi\ v * \mathsf{tok}\ \gamma_s}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\mathsf{tok}\ \gamma_s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

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

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. Use invariant with disjunct for each state
3. Determine resource ownership of each state
4. Encode send/recv transition as transferring a token to the invariant



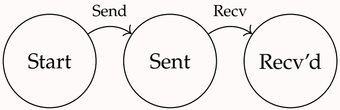
$$\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi \triangleq (\underbrace{c \mapsto \mathtt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \mathtt{Some}\, v * \Phi\ v * \mathsf{tok}\ \gamma_s}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\mathsf{tok}\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

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

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. Use invariant with disjunct for each state
3. Determine resource ownership of each state
4. Encode send/recv transition as transferring a token to the invariant
5. Give sender/receiver their token and access to the invariant



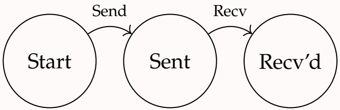
$$\mathsf{chan\_inv}\ \gamma_s\ \gamma_r\ c\ \Phi \triangleq (\underbrace{c \mapsto \mathtt{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v.\, c \mapsto \mathtt{Some}\, v * \Phi\, v * \mathsf{tok}\ \gamma_s}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\mathsf{tok}\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

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

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. Use invariant with disjunct for each state
3. Determine resource ownership of each state
4. Encode send/recv transition as transferring a token to the invariant
5. Give sender/receiver their token and access to the invariant



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

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

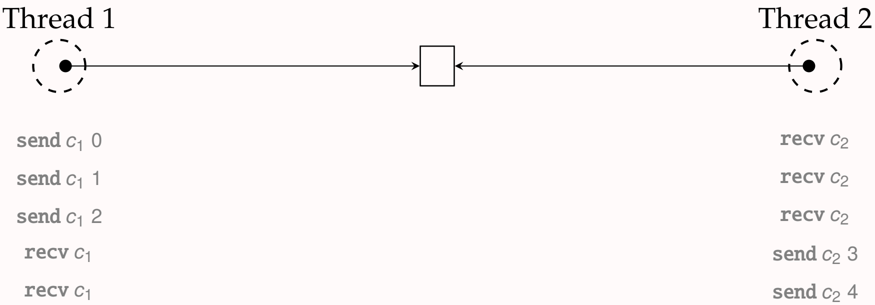
Layer 2: Functional Session Channels

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives:

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

Emerging polarised bi-directional linked list:

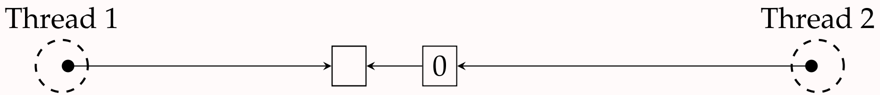


Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives:

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

Emerging polarised bi-directional linked list:



$\texttt{send}\ c_1\ 0$	$\texttt{recv}\ c_2$
$\texttt{send}\ c_1\ 1$	$\texttt{recv}\ c_2$
$\texttt{send}\ c_1\ 2$	$\texttt{recv}\ c_2$
$\texttt{recv}\ c_1$	$\texttt{send}\ c_2\ 3$
$\texttt{recv}\ c_1$	$\texttt{send}\ c_2\ 4$

Layer 2: Functional Session Channels (Implementation)

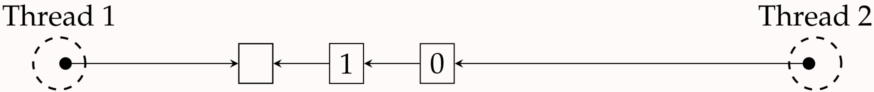
Functional session channel primitives:

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

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

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

Emerging polarised bi-directional linked list:



$\mathtt{send}\ c_1\ 0$

$\mathtt{send}\ c_1\ 1$

$\mathtt{send}\ c_1\ 2$

$\mathtt{recv}\ c_1$

$\mathtt{recv}\ c_1$

$\mathtt{recv}\ c_2$

$\mathtt{recv}\ c_2$

$\mathtt{recv}\ c_2$

$\mathtt{send}\ c_2\ 3$

$\mathtt{send}\ c_2\ 4$

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives:

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

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

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

Emerging polarised bi-directional linked list:

Thread 1	Thread 2
$\mathtt{send}\ c_1\ 0$	$\mathtt{recv}\ c_2$
$\mathtt{send}\ c_1\ 1$	$\mathtt{recv}\ c_2$
$\mathtt{send}\ c_1\ 2$	$\mathtt{recv}\ c_2$
$\mathtt{recv}\ c_1$	$\mathtt{send}\ c_2\ 3$
$\mathtt{recv}\ c_1$	$\mathtt{send}\ c_2\ 4$

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives:

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

Emerging polarised bi-directional linked list:

Thread 1	Thread 2
$\texttt{send}\ c_1\ 0$	$\texttt{recv}\ c_2$
$\texttt{send}\ c_1\ 1$	$\texttt{recv}\ c_2$
$\texttt{send}\ c_1\ 2$	$\texttt{recv}\ c_2$
$\texttt{recv}\ c_1$	$\texttt{send}\ c_2\ 3$
$\texttt{recv}\ c_1$	$\texttt{send}\ c_2\ 4$

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives:

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

Emerging polarised bi-directional linked list:

$\mathbf{send}\ c_1\ 0$	$\mathbf{recv}\ c_2$
$\mathbf{send}\ c_1\ 1$	$\mathbf{recv}\ c_2$
$\mathbf{send}\ c_1\ 2$	$\mathbf{recv}\ c_2$
$\mathbf{recv}\ c_1$	$\mathbf{send}\ c_2\ 3$
$\mathbf{recv}\ c_1$	$\mathbf{send}\ c_2\ 4$

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives:

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

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

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

Emerging polarised bi-directional linked list:

Thread 1 | Thread 2

Thread 1	Thread 2
$\texttt{send}\ c_1\ 0$	$\texttt{recv}\ c_2$
$\texttt{send}\ c_1\ 1$	$\texttt{recv}\ c_2$
$\texttt{send}\ c_1\ 2$	$\texttt{recv}\ c_2$
$\texttt{recv}\ c_1$	$\texttt{send}\ c_2\ 3$
$\texttt{recv}\ c_1$	$\texttt{send}\ c_2\ 4$

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives:

$$\mathbf{new}\ () \triangleq \mathbf{new1}\ ()$$

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

$$\mathbf{recv}\ c \triangleq \mathbf{recv1}\ c$$

Emerging polarised bi-directional linked list:

$\mathbf{send}\ c_1\ 0$	$\mathbf{recv}\ c_2$
$\mathbf{send}\ c_1\ 1$	$\mathbf{recv}\ c_2$
$\mathbf{send}\ c_1\ 2$	$\mathbf{recv}\ c_2$
$\mathbf{recv}\ c_1$	$\mathbf{send}\ c_2\ 3$
$\mathbf{recv}\ c_1$	$\mathbf{send}\ c_2\ 4$

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives:

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

Emerging polarised bi-directional linked list:

$\mathtt{send}\ c_1\ 0$	$\mathtt{recv}\ c_2$
$\mathtt{send}\ c_1\ 1$	$\mathtt{recv}\ c_2$
$\mathtt{send}\ c_1\ 2$	$\mathtt{recv}\ c_2$
$\mathtt{recv}\ c_1$	$\mathtt{send}\ c_2\ 3$
$\mathtt{recv}\ c_1$	$\mathtt{send}\ c_2\ 4$

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives:

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

Emerging polarised bi-directional linked list:

send c_1 0	**recv** c_2
send c_1 1	**recv** c_2
send c_1 2	**recv** c_2
recv c_1	**send** c_2 3
recv c_1	**send** c_2 4

Layer 2: Functional Session Channels (Implementation)

Functional session channel primitives:

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

Emerging polarised bi-directional linked list:

Thread 1	Thread 2
$\textbf{send}\ c_1\ 0$	$\textbf{recv}\ c_2$
$\textbf{send}\ c_1\ 1$	$\textbf{recv}\ c_2$
$\textbf{send}\ c_1\ 2$	$\textbf{recv}\ c_2$
$\textbf{recv}\ c_1$	$\textbf{send}\ c_2\ 3$
$\textbf{recv}\ c_1$	$\textbf{send}\ c_2\ 4$

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:

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

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:

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

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:

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

Functional session channel primitives:

$$\mathbf{new}\ () \triangleq \mathbf{new1}\ ()$$

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

$$\mathbf{recv}\ c \triangleq \mathbf{recv1}\ c$$

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:

$$\mathbf{new}\ () \triangleq \mathbf{new1}\ ()$$

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

$$\mathbf{recv}\ c \triangleq \mathbf{recv1}\ c$$

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:

$$\mathbf{new}\ () \triangleq \mathbf{new1}\ ()$$

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

$$\mathbf{recv}\ c \triangleq \mathbf{recv1}\ c$$

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:

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

Dependent session protocols:

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

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:

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

Layer 2: Functional Session Channels (Protocols and Specifications)

Functional session channel primitives:

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

Dependent session protocols:

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

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:

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

Specifications:

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

Functional session channel primitives:

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

Dependent session protocols:

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

Specifications:

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

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:

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

Specifications:

$$\{\mathsf{True}\}\;\mathtt{new}\,()\;\{c.\, c \rightarrowtail p * c \rightarrowtail \overline{p}\}$$

$$\{c \rightarrowtail (\mathbf{!}\,(x:\tau)\,\langle w\rangle\{P\}.\,p) * P\;t\}\;\mathtt{send}\,c\,(w\;t)\;\{c'.\, c' \rightarrowtail p\;t\}$$

$$\{c \rightarrowtail (\mathbf{?}\,(x:\tau)\,\langle w\rangle\{P\}.\,p)\}\;\mathtt{recv}\,c\;\{(v, c').\, \exists(x:\tau).\, v = (w\;x) * P\;x * c' \rightarrowtail p\;x\}$$

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:

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

Specifications:

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

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 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 definitions:

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

Unfolding the definitions yield the following nesting:

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

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

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

Recall the definitions:

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

Unfolding the definitions yield the following nesting:

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

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

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

Recall the definitions:

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

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

Unfolding the definitions yield the following nesting:

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

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

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

Recall the definitions:

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

Unfolding the definitions yield the following nesting:

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

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

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

Recall the definitions:

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

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

Unfolding the definitions yield the following nesting:

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

Nested invariants are *supported by Iris*

Layer 3: Imperative Channels

Layer 3: Imperative Channels (Motivation and Implementation)

Functional channels are inconvenient:

$$\ldots \, \mathtt{let}\, c = \mathtt{send}\, c\, \ell\, \mathtt{in}\, \mathtt{recv}\, c; \ldots$$

We instead want:

$$\ldots \, c.\mathtt{send}(\ell); c.\mathtt{recv}(); \ldots$$

Layer 3: Imperative Channels (Motivation and Implementation)

Functional channels are inconvenient:

$$\ldots \texttt{let}\, c = \texttt{send}\, c\ \ell\ \texttt{in}\, \texttt{recv}\, c; \ldots$$

We instead want:

$$\ldots c.\texttt{send}(\ell); c.\texttt{recv}(); \ldots$$

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:

$$\ldots \mathtt{let}\, c = \mathtt{send}\, c\; \ell\; \mathtt{in}\, \mathtt{recv}\, c; \ldots$$

We instead want:

$$\ldots c.\mathtt{send}(\ell); c.\mathtt{recv}(); \ldots$$

Solution: Imperative channels

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

With this we can write the program from the introduction:

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

Layer 3: Imperative Channels (Specifications and Proof)

Points-to:

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

Layer 3: Imperative Channels (Specifications and Proof)

Points-to:

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

Actris Specifications:

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

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

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

Layer 3: Imperative Channels (Specifications and Proof)

Points-to:

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

Actris Specifications:

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

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

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

Proof of specifications is trivial reasoning about references

Layer 3: Imperative Channels (Example and Proof)

Program from introduction:

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

Protocol:

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

Layer 3: Imperative Channels (Example and Proof)

Program from introduction:

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

Protocol:

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

We can now prove this via specifications fully derived during this talk!

Additional Features of MiniActris

Other Features of MiniActris

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

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

Channel deallocation: traditional & new (**`send_close`**)

Sending channels as messages, integration with Iris, . . .

Other 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 & new (**send_close**)

Sending channels as messages, integration with Iris, . . .

Everything mechanized in less than 1000 lines of Coq!

$\mathsf{prog\_single} \triangleq$
  $\mathbf{let}\ c = \mathbf{new1}\ ()\ \mathbf{in}$
  $\mathbf{fork}\ \{\mathbf{let}\ l = \mathbf{ref}\ 42\ \mathbf{in}\ \mathbf{send1}\ c\ l\};$
  $\mathbf{assert}(!(\mathbf{recv1}\ c) = 42)$

Click!

```
Definition prog_single : val :=
  λ: "<>",
    let: "c" := new1 #() in
    Fork (let: "l" := ref #42 in send1 "c" "l");;
    let: "l" := recv1 "c" in assert: (!"l" = #42).
```

Concluding Remarks

Comparison with Actris

MiniActris

This work

(ICFP'23)

Asynchronous channels

Dependent session protocols

Iris separation logic

Channels over channels

Recursive protocols

Channel deallocation

Variance subprotocols

Comparison with Actris

MiniActris

This work

(ICFP'23)

Asynchronous channels

Dependent session protocols

Iris separation logic

Channels over channels

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 over channels

Recursive protocols

Channel deallocation

Variance subprotocols

Asynchronous subprotocols

Ghost theory

Actris 1.0

Hinrichsen, Bengtson, Krebbers

(POPL'20)

Actris 2.0

Hinrichsen, Bengtson, Krebbers

(LMCS'22)

Comparison with Actris

MiniActris

This work

(ICFP'23)

Asynchronous channels

Dependent session protocols

Iris separation logic

Channels over channels

Recursive protocols

n

s

cols

Actris 1.0

Hinrichsen, Bengtson, Krebbers

(POPL'20)

Actris 2.0

Hinrichsen, Bengtson, Krebbers

(LMCS'22)

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

Ghost theory

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

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

- Three layers: one-shot $\rightarrow$ functional $\rightarrow$ imperative
- Simple soundness proof with nested invariants
- Abundance of protocol features
- Mechanized in 1000 lines of Coq

Suitable as an exercise in separation logic courses?

- One-shot channels: *suitable*
- Dependent session protocols: *nested one-shot protocols*

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