The Actris Ghost Theory:
Session Type-Based Ghost Theory for Reasoning about Reliable Communication

Jonas Kastberg Hinrichsen, Aarhus University

joint work with
Jesper Bengtson, IT University of Copenhagen
Robbert Krebbers, Radboud University

2. May 2022
Iris Workshop'22

Reliable communication

Reliable communication has a lot of applications

Reliable communication

Reliable communication has a lot of applications

- Shared memory message passing (Go)

Reliable communication

Reliable communication has a lot of applications

- Shared memory message passing (Go)
- Distributed networks (TCP)

Reliable communication

Reliable communication has a lot of applications

- Shared memory message passing (Go)
- Distributed networks (TCP)

Communication which assumes that:

- Messages are never dropped, duplicated, or arrive out of order

Reliable communication

Reliable communication has a lot of applications

- Shared memory message passing (Go)
- Distributed networks (TCP)

Communication which assumes that:

- Messages are never dropped, duplicated, or arrive out of order

We additionally assume:

- Binary - communication is between two participants

Reliable communication

Reliable communication has a lot of applications

- Shared memory message passing (Go)
- Distributed networks (TCP)

Communication which assumes that:

- Messages are never dropped, duplicated, or arrive out of order

We additionally assume:

- Binary - communication is between two participants

Shared memory message passing primitives (in HeapLang)

`new_chan ()`, `send c v`, `recv c`

Reliable communication

Reliable communication has a lot of applications

- Shared memory message passing (Go)
- Distributed networks (TCP)

Communication which assumes that:

- Messages are never dropped, duplicated, or arrive out of order

We additionally assume:

- Binary - communication is between two participants

Shared memory message passing primitives (in HeapLang)

$\texttt{new\_chan}\ ()$, $\texttt{send c v}$, $\texttt{recv c}$

Example Program:

$$\begin{array}{ll}
\texttt{let}\ (c, c') := \texttt{new_chan}\ ()\ \texttt{in} & \\
\texttt{fork}\ \{\texttt{let}\ x := \texttt{recv}\ c'\ \texttt{in}\ \texttt{send}\ c'\ (x+2)\}\,; & \text{// Service thread} \\
\texttt{send}\ c\ 40;\ \texttt{recv}\ c & \text{// Client thread}
\end{array}$$

Session types

Syntax

$$\begin{aligned} A ::= {} & \mathbf{Z} \mid \mathbf{B} \mid \mathbf{1} \mid \\ & \texttt{chan}\ S \mid \ldots \end{aligned}$$

Session types

Syntax

$$
\begin{aligned}
A &::= \mathbf{Z} \mid \mathbf{B} \mid \mathbf{1} \mid \\
&\quad\ \ \texttt{chan}\ S \mid \dots
\end{aligned}
$$

$$
\begin{aligned}
S &::= \mathbf{!}A.\,S \quad \mid \\
&\quad\ \ \mathbf{?}A.\,S \quad \mid \\
&\quad\ \ \texttt{end} \quad\ \mid \dots
\end{aligned}
$$

Session types

Syntax

$$
\begin{aligned}
A ::= {} & \mathbf{Z} \mid \mathbf{B} \mid \mathbf{1} \mid \\
& \mathtt{chan}\ S \mid \ldots
\end{aligned}
$$

$$
\begin{aligned}
S ::= {} & \mathbf{!}A.\,S \quad \mid \\
& \mathbf{?}A.\,S \quad \mid \\
& \mathtt{end} \quad \mid \ldots
\end{aligned}
$$

Example

$$\mathtt{chan}\ (\mathbf{!Z}.\,\mathbf{?Z}.\,\mathtt{end})$$

Session types

Syntax

$$A ::= \mathbf{Z} \mid \mathbf{B} \mid \mathbf{1} \mid \texttt{chan}\ S \mid \ldots$$

$$S ::= \mathbf{!}A.\, S \mid \mathbf{?}A.\, S \mid \texttt{end} \mid \ldots$$

Example

$$\texttt{chan}\ (\mathbf{!Z}.\, \mathbf{?Z}.\, \texttt{end})$$

Usage

$$c : \texttt{chan}\ S$$

Syntax

$$
\begin{aligned}
A &::= \mathbf{Z} \mid \mathbf{B} \mid \mathbf{1} \mid \\
&\quad\ \ \texttt{chan}\ S \mid \ldots
\end{aligned}
$$

$$
\begin{aligned}
S ::= &\ \mathbf{!}A.\,S & \mid \\
&\ \mathbf{?}A.\,S & \mid \\
&\ \texttt{end} & \mid \ldots
\end{aligned}
$$

Duality

$$
\begin{aligned}
\overline{\mathbf{!}A.\,S} &= \mathbf{?}A.\,\overline{S} \\
\overline{\mathbf{?}A.\,S} &= \mathbf{!}A.\,\overline{S} \\
\overline{\texttt{end}} &= \texttt{end}
\end{aligned}
$$

Example

$$\texttt{chan}\ (\mathbf{!Z}.\,\mathbf{?Z}.\,\texttt{end})$$

Usage

$$c : \texttt{chan}\ S$$

Session types

Syntax

$$A ::= \mathbf{Z} \mid \mathbf{B} \mid \mathbf{1} \mid \texttt{chan}\ S \mid \ldots$$

$$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \texttt{end} \mid \ldots$$

Example

$$\texttt{chan}\ (\mathbf{!Z}.\,\mathbf{?Z}.\,\texttt{end})$$

Usage

$$c : \texttt{chan}\ S$$

Duality

$$\begin{aligned} \overline{\mathbf{!}A.\,S} &= \mathbf{?}A.\,\overline{S} \\ \overline{\mathbf{?}A.\,S} &= \mathbf{!}A.\,\overline{S} \\ \overline{\texttt{end}} &= \texttt{end} \end{aligned}$$

Rules (for shared memory message passing)

$$\Gamma \vdash \texttt{new\_chan}\ () : \texttt{chan}\ S \times \texttt{chan}\ \overline{S} \dashv \Gamma$$

Session types

Syntax

$$A ::= \mathbf{Z} \mid \mathbf{B} \mid \mathbf{1} \mid \texttt{chan}\ S \mid \ldots$$

$$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \texttt{end} \mid \ldots$$

Example

$$\texttt{chan}\ (\mathbf{!Z}.\,\mathbf{?Z}.\,\texttt{end})$$

Usage

$$c : \texttt{chan}\ S$$

Duality

$$\begin{aligned} \overline{\mathbf{!}A.\,S} &= \mathbf{?}A.\,\overline{S} \\ \overline{\mathbf{?}A.\,S} &= \mathbf{!}A.\,\overline{S} \\ \overline{\texttt{end}} &= \texttt{end} \end{aligned}$$

Rules (for shared memory message passing)

$$\Gamma \vdash \texttt{new\_chan}\ () : \texttt{chan}\ S \times \texttt{chan}\ \overline{S} \dashv \Gamma$$

$$\Gamma, x : \texttt{chan}\ (\mathbf{!}A.\,S), y : A \vdash \texttt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \texttt{chan}\ S$$

Session types

Syntax

$$A ::= \mathbf{Z} \mid \mathbf{B} \mid \mathbf{1} \mid \texttt{chan}\ S \mid \ldots$$

$$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \texttt{end} \mid \ldots$$

Example

$\texttt{chan}\ (\mathbf{!Z}.\,\mathbf{?Z}.\,\texttt{end})$

Usage

$c : \texttt{chan}\ S$

Duality

$$\begin{aligned} \overline{\mathbf{!}A.\,S} &= \mathbf{?}A.\,\overline{S} \\ \overline{\mathbf{?}A.\,S} &= \mathbf{!}A.\,\overline{S} \\ \overline{\texttt{end}} &= \texttt{end} \end{aligned}$$

Rules (for shared memory message passing)

$$\Gamma \vdash \texttt{new\_chan}\ () : \texttt{chan}\ S \times \texttt{chan}\ \overline{S} \dashv \Gamma$$
$$\Gamma, x : \texttt{chan}\ (\mathbf{!}A.\,S), y : A \vdash \texttt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \texttt{chan}\ S$$
$$\Gamma, x : \texttt{chan}\ (\mathbf{?}A.\,S) \vdash \texttt{recv}\ x : A \dashv \Gamma, x : \texttt{chan}\ S$$

Session types

Syntax

$$A ::= \mathbf{Z} \mid \mathbf{B} \mid \mathbf{1} \mid \texttt{chan}\ S \mid \ldots$$

$$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \texttt{end} \mid \ldots$$

Example

$$\texttt{chan}\ (\mathbf{!Z}.\,\mathbf{?Z}.\,\texttt{end})$$

Usage

$$c : \texttt{chan}\ S$$

Duality

$$\begin{aligned} \overline{\mathbf{!}A.\,S} &= \mathbf{?}A.\,\overline{S} \\ \overline{\mathbf{?}A.\,S} &= \mathbf{!}A.\,\overline{S} \\ \overline{\texttt{end}} &= \texttt{end} \end{aligned}$$

Rules (for shared memory message passing)

$$\Gamma \vdash \texttt{new\_chan}\ () : \texttt{chan}\ S \times \texttt{chan}\ \overline{S} \dashv \Gamma$$
$$\Gamma, x : \texttt{chan}\ (\mathbf{!}A.\,S), y : A \vdash \texttt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \texttt{chan}\ S$$
$$\Gamma, x : \texttt{chan}\ (\mathbf{?}A.\,S) \vdash \texttt{recv}\ x : A \dashv \Gamma, x : \texttt{chan}\ S$$

Example program (service thread)

$$\begin{aligned} &\lambda c.\, \texttt{let}\ x := \texttt{recv}\ c\ \texttt{in} \\ &\quad \texttt{send}\ c\ (x + 2) \end{aligned}$$

Session types

Syntax

$$A ::= \mathbf{Z} \mid \mathbf{B} \mid \mathbf{1} \mid \texttt{chan}\ S \mid \ldots$$

$$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \texttt{end} \mid \ldots$$

Example

$$\texttt{chan}\ (\mathbf{!Z}.\,\mathbf{?Z}.\,\texttt{end})$$

Usage

$$c : \texttt{chan}\ S$$

Duality

$$\begin{aligned} \overline{\mathbf{!}A.\,S} &= \mathbf{?}A.\,\overline{S} \\ \overline{\mathbf{?}A.\,S} &= \mathbf{!}A.\,\overline{S} \\ \overline{\texttt{end}} &= \texttt{end} \end{aligned}$$

Rules (for shared memory message passing)

$$\Gamma \vdash \texttt{new\_chan}\ () : \texttt{chan}\ S \times \texttt{chan}\ \overline{S} \dashv \Gamma$$

$$\Gamma, x : \texttt{chan}\ (\mathbf{!}A.\,S), y : A \vdash \texttt{send}\ x\ y : \mathbf{1} \dashv \Gamma, x : \texttt{chan}\ S$$

$$\Gamma, x : \texttt{chan}\ (\mathbf{?}A.\,S) \vdash \texttt{recv}\ x : A \dashv \Gamma, x : \texttt{chan}\ S$$

Example program (service thread)

$$\begin{aligned} \Gamma \vdash \lambda c.\ & \texttt{let}\ x := \texttt{recv}\ c\ \texttt{in} \\ & \texttt{send}\ c\ (x+2) : \texttt{chan}\ (\mathbf{?Z}.\,\mathbf{!Z}.\,\texttt{end}) \multimap \mathbf{1} \dashv \Gamma \end{aligned}$$

Example program - via session types

Example program:

$$
\begin{array}{ll}
\texttt{let}\,(c, c') := \texttt{new\_chan}\ ()\ \texttt{in} & \\
\texttt{fork}\ \{\texttt{let}\, x := \texttt{recv}\ c'\ \texttt{in}\, \texttt{send}\ c'\ (x + 2)\}\,; & //\ \text{Service thread} \\
\texttt{send}\ c\ 40;\ \texttt{recv}\ c & //\ \text{Client thread}
\end{array}
$$

Example program - via session types

Example program:

$$\texttt{let}\,(c, c') := \texttt{new\_chan}\;()\;\texttt{in}$$
$$\texttt{fork}\;\{\texttt{let}\,x := \texttt{recv}\;c'\;\texttt{in}\,\texttt{send}\;c'\;(x+2)\}\,; \quad // \text{ Service thread}$$
$$\texttt{send}\;c\;40;\;\texttt{recv}\;c \quad // \text{ Client thread}$$

Session types:

$$c \;:\; \texttt{chan}\;(\mathbf{!Z}.\,\mathbf{?Z}.\,\texttt{end}) \qquad \texttt{and}$$
$$c' \;:\; \texttt{chan}\;(\mathbf{?Z}.\,\mathbf{!Z}.\,\texttt{end})$$

Example program - via session types

Example program:

$$\texttt{let}\,(c, c') := \texttt{new\_chan}\;()\;\texttt{in}$$
$$\texttt{fork}\;\{\texttt{let}\,x := \texttt{recv}\;c'\;\texttt{in}\,\texttt{send}\;c'\;(x+2)\}\,; \qquad \text{// Service thread}$$
$$\texttt{send}\;c\;40;\;\texttt{recv}\;c \qquad \text{// Client thread}$$

Session types:

$$c \;:\; \texttt{chan}\;(\mathbf{!Z}.\,\mathbf{?Z}.\,\texttt{end}) \qquad \texttt{and}$$
$$c' \;:\; \texttt{chan}\;(\mathbf{?Z}.\,\mathbf{!Z}.\,\texttt{end})$$

Properties obtained:

☑ Program does not crash

Example program - via session types

Example program:

$$\begin{array}{ll} \texttt{let}\,(c, c') := \texttt{new\_chan}\;()\;\texttt{in} & \\ \texttt{fork}\;\{\texttt{let}\,x := \texttt{recv}\;c'\;\texttt{in}\,\texttt{send}\;c'\;(x+2)\}\,; & \text{// Service thread} \\ \texttt{send}\;c\;40;\;\texttt{recv}\;c & \text{// Client thread} \end{array}$$

Session types:

$$\begin{aligned} c &: \texttt{chan}\;(\mathbf{!Z}.\,\mathbf{?Z}.\,\texttt{end}) \qquad \text{and} \\ c' &: \texttt{chan}\;(\mathbf{?Z}.\,\mathbf{!Z}.\,\texttt{end}) \end{aligned}$$

Properties obtained:

- ☑ Program does not crash
- ☒ Program is correct (returns 42)

Problems

1. **Lack of expressivity in session types**
   - Restricted to decidable fragment
   - Does not guarantee functional correctness

Problems

1. **Lack of expressivity in session types**
   - Restricted to decidable fragment
   - Does not guarantee functional correctness
2. **Lack of generality with respect to the underlying implementation**
   - Communication is assumed to be reliable at the level of the operational semantics
   - Does not readily integrate with reliable communication that is implemented

Problems

1. **Lack of expressivity in session types**
   - Restricted to decidable fragment
   - Does not guarantee functional correctness
2. **Lack of generality with respect to the underlying implementation**
   - Communication is assumed to be reliable at the level of the operational semantics
   - Does not readily integrate with reliable communication that is implemented
3. **Lack of mechanisation results of session type-based systems**
   - Few results of simpler systems
   - No results of systems that combine features such as recursion and subtyping

Key Idea

Protocols akin to **session types** for reasoning in the **Iris concurrent separation logic**

Key Idea

Protocols akin to **session types** for reasoning in the **Iris concurrent separation logic**

Session types

- Modular verification of channel endpoints
- Ensures safety

Key Idea

Protocols akin to **session types** for reasoning in the **Iris concurrent separation logic**

Session types

- Modular verification of channel endpoints
- Ensures safety

Iris concurrent separation logic

- Logic for reasoning about concurrent programs
- Ensures functional correctness

Key Idea

Protocols akin to **session types** for reasoning in the **Iris concurrent separation logic**

Session types

- Modular verification of channel endpoints
- Ensures safety

Iris concurrent separation logic

- Logic for reasoning about concurrent programs
- Ensures functional correctness
- General purpose ghost state mechanisms
  - Implementation-agnostic logical state and its transitions

Key Idea

Protocols akin to **session types** for reasoning in the **Iris concurrent separation logic**

Session types

- Modular verification of channel endpoints
- Ensures safety

Iris concurrent separation logic

- Logic for reasoning about concurrent programs
- Ensures functional correctness
- General purpose ghost state mechanisms
  - Implementation-agnostic logical state and its transitions
- Full mechanisation in Coq

Contributions

Actris: A framework for proving *functional correctness* of programs that implement and use the *reliable communication* paradigm

Contributions

Actris: A framework for proving *functional correctness* of programs that implement and use the *reliable communication* paradigm

1. Introducing *dependent separation protocols*
   - Higher-order separation logic session protocols for specifying functional behaviour
     - Step-indexed recursion
     - Subprotocols inspired by asynchronous session subtyping

Contributions

Actris: A framework for proving *functional correctness* of programs that implement and use the *reliable communication* paradigm

1. Introducing *dependent separation protocols*
   - Higher-order separation logic session protocols for specifying functional behaviour
     - Step-indexed recursion
     - Subprotocols inspired by asynchronous session subtyping
2. The *Actris* rules (for HeapLang)
   - Implementation-specific session type-style rules for verifying programs that use reliable communication

Contributions

Actris: A framework for proving *functional correctness* of programs that implement and use the *reliable communication* paradigm

1. Introducing *dependent separation protocols*
   - Higher-order separation logic session protocols for specifying functional behaviour
     - Step-indexed recursion
     - Subprotocols inspired by asynchronous session subtyping
2. The *Actris* rules (for HeapLang)
   - Implementation-specific session type-style rules for verifying programs that use reliable communication
3. The *Actris Ghost Theory*
   - Implementation-agnostic framework for specifying and proving implementation-specific Actris rules

Contributions

Actris: A framework for proving *functional correctness* of programs that implement and use the *reliable communication* paradigm

1. Introducing *dependent separation protocols*
   - Higher-order separation logic session protocols for specifying functional behaviour
     - Step-indexed recursion
     - Subprotocols inspired by asynchronous session subtyping
2. The *Actris* rules (for HeapLang)
   - Implementation-specific session type-style rules for verifying programs that use reliable communication
3. The *Actris Ghost Theory*
   - Implementation-agnostic framework for specifying and proving implementation-specific Actris rules
4. A full mechanisation of Actris on top of Iris in Coq
   - With tactic support
   - https://gitlab.mpi-sws.org/iris/actris/

Contributions

Actris: A framework for proving *functional correctness* of programs that implement and use the *reliable communication* paradigm

1. Introducing *dependent separation protocols* [POPL'20]
   - Higher-order separation logic session protocols for specifying functional behaviour
     - Step-indexed recursion
     - Subprotocols inspired by asynchronous session subtyping
2. The *Actris* rules (for HeapLang) [POPL'20]
   - Implementation-specific session type-style rules for verifying programs that use reliable communication
3. The *Actris Ghost Theory*
   - Implementation-agnostic framework for specifying and proving implementation-specific Actris rules
4. A full mechanisation of Actris on top of Iris in Coq [POPL'20]
   - With tactic support
   - https://gitlab.mpi-sws.org/iris/actris/

Contributions

Actris: A framework for proving *functional correctness* of programs that implement and use the *reliable communication* paradigm

1. Introducing *dependent separation protocols* [POPL'20]
   - Higher-order separation logic session protocols for specifying functional behaviour
     - Step-indexed recursion
     - Subprotocols inspired by asynchronous session subtyping [LMCS'22]
2. The *Actris* rules (for HeapLang) [POPL'20]
   - Implementation-specific session type-style rules for verifying programs that use reliable communication
3. The *Actris Ghost Theory* [LMCS'22]
   - Implementation-agnostic framework for specifying and proving implementation-specific Actris rules
4. A full mechanisation of Actris on top of Iris in Coq [POPL'20] [LMCS'22]
   - With tactic support
   - https://gitlab.mpi-sws.org/iris/actris/

1. Dependent separation protocols
2. Actris Rules
3. Actris Ghost Theory
4. Mechanisation of Actris

Dependent separation protocols

Session type-inspired protocols for functional correctness

Dependent separation protocols

Session type-inspired protocols for functional correctness:

- Exchanges of: logical variables ($\vec{x}:\vec{\tau}$)

Dependent separation protocols

Session type-inspired protocols for functional correctness:

- Exchanges of: logical variables ($\vec{x} : \vec{\tau}$), physical values (v)

Dependent separation protocols

Session type-inspired protocols for functional correctness:

- Exchanges of: logical variables ($\vec{x}:\vec{\tau}$), physical values (v), propositions (P)

Dependent separation protocols

Session type-inspired protocols for functional correctness:

- Exchanges of: logical variables ($\vec{x}:\vec{\tau}$), physical values (v), propositions (P)

	Dependent separation protocols	**Session types**
Syntax	$\mathit{prot} ::= \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} \mid \mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} \mid \mathbf{end}$	$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \texttt{end} \mid \ldots$

Dependent separation protocols

Session type-inspired protocols for functional correctness:

- Exchanges of: logical variables ($\vec{x}:\vec{\tau}$), physical values (v), propositions (P)

	Dependent separation protocols	**Session types**
Syntax	$prot ::= \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot \mid \mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot \mid \mathbf{end}$	$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \texttt{end} \mid \ldots$
Example	$\mathbf{!}\,(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,\mathbf{?}(y:\mathbb{Z})\,\langle y\rangle\{y=(x+2)\}.\,\mathbf{end}$	$\mathbf{!Z}.\,\mathbf{?Z}.\,\texttt{end}$

Dependent separation protocols

Session type-inspired protocols for functional correctness:

- Exchanges of: logical variables ($\vec{x}:\vec{\tau}$), physical values (v), propositions (P)

	Dependent separation protocols	Session types
Syntax	$prot ::= \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot \mid \mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot \mid \mathbf{end}$	$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \texttt{end} \mid \ldots$
Example	$\mathbf{!}\,(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,\mathbf{?}(y:\mathbb{Z})\,\langle y\rangle\{y=(x+2)\}.\,\mathbf{end}$	$\mathbf{!Z}.\,\mathbf{?Z}.\,\texttt{end}$
Duality	$\overline{\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot} = \mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\overline{prot}$ $\overline{\mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot} = \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\overline{prot}$ $\overline{\mathbf{end}} = \mathbf{end}$	$\overline{\mathbf{!}A.\,S} = \mathbf{?}A.\,\overline{S}$ $\overline{\mathbf{?}A.\,S} = \mathbf{!}A.\,\overline{S}$ $\overline{\texttt{end}} = \texttt{end}$

Dependent separation protocols

Session type-inspired protocols for functional correctness:

- Exchanges of: logical variables ($\vec{x}:\vec{\tau}$), physical values (v), propositions (P)
- Dependent: the variables $\vec{x}:\vec{\tau}$ bind into v, P, and *prot*

	Dependent separation protocols	**Session types**
Syntax	$\mathit{prot} ::= \,!\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} \mid ?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} \mid \mathbf{end}$	$S ::= \,!A.\,S \mid ?A.\,S \mid \mathtt{end} \mid \ldots$
Example	$!\,(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,?(y:\mathbb{Z})\,\langle y\rangle\{y=(x+2)\}.\,\mathbf{end}$	$!\mathbf{Z}.\,?\mathbf{Z}.\,\mathtt{end}$
Duality	$\overline{!\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}} = ?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\overline{\mathit{prot}}$ $\overline{?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}} = \,!\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\overline{\mathit{prot}}$ $\overline{\mathbf{end}} = \mathbf{end}$	$\overline{!A.\,S} = ?A.\,\overline{S}$ $\overline{?A.\,S} = \,!A.\,\overline{S}$ $\overline{\mathtt{end}} = \mathtt{end}$

Dependent separation protocols

Session type-inspired protocols for functional correctness:

- Exchanges of: logical variables ($\vec{x}:\vec{\tau}$), physical values (v), propositions (P)
- Dependent: the variables $\vec{x}:\vec{\tau}$ bind into v, P, and *prot*
- First class citizens of Iris (COFEs): higher-order, impredicativity, recursion

	Dependent separation protocols	**Session types**
Syntax	$\mathit{prot} ::= \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} \mid \mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} \mid \mathbf{end}$	$S ::= \mathbf{!}A.\,S \mid \mathbf{?}A.\,S \mid \mathtt{end} \mid \ldots$
Example	$\mathbf{!}\,(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,\mathbf{?}(y:\mathbb{Z})\,\langle y\rangle\{y=(x+2)\}.\,\mathbf{end}$	$\mathbf{!Z}.\,\mathbf{?Z}.\,\mathtt{end}$
Duality	$\overline{\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}} = \mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\overline{\mathit{prot}}$ $\overline{\mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}} = \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\overline{\mathit{prot}}$ $\overline{\mathbf{end}} = \mathbf{end}$	$\overline{\mathbf{!}A.\,S} = \mathbf{?}A.\,\overline{S}$ $\overline{\mathbf{?}A.\,S} = \mathbf{!}A.\,\overline{S}$ $\overline{\mathtt{end}} = \mathtt{end}$

1. Dependent separation protocols
2. Actris Rules
3. Actris Ghost Theory
4. Mechanisation of Actris

Actris Rules (for HeapLang)

	Actris	**Session types**
Usage	$c \rightarrowtail \mathit{prot}$	$c : \texttt{chan}\ S$

Actris Rules (for HeapLang)

	Actris	**Session types**
Usage	$c \rightarrowtail \mathit{prot}$	$c\ :\ \texttt{chan}\ S$
New	$\{\mathsf{True}\}$ $\quad \texttt{new\_chan}\ ()$ $\{(c, c').\ c \rightarrowtail \mathit{prot} * c' \rightarrowtail \overline{\mathit{prot}}\}$	$\Gamma \vdash \texttt{new\_chan}\ () : \texttt{chan}\ S \times \texttt{chan}\ \overline{S} \dashv \Gamma$

Actris Rules (for HeapLang)

	Actris	**Session types**
Usage	$c \rightarrowtail \mathit{prot}$	$c : \texttt{chan}\ S$
New	$\{\mathrm{True}\}$ $\quad\texttt{new\_chan}\ ()$ $\{(c, c').\ c \rightarrowtail \mathit{prot} * c' \rightarrowtail \overline{\mathit{prot}}\}$	$\Gamma \vdash \texttt{new\_chan}\ () : \texttt{chan}\ S \times \texttt{chan}\ \overline{S} \dashv \Gamma$
Send	$\{c \rightarrowtail \mathbf{!}\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\ \mathit{prot} * P[\vec{t}/\vec{x}]\}$ $\quad\texttt{send}\ c\ (v[\vec{t}/\vec{x}])$ $\{c \rightarrowtail \mathit{prot}[\vec{t}/\vec{x}]\}$	$\Gamma, x : \texttt{chan}\ (\mathbf{!}A.\ S), y : A \vdash \texttt{send}\ x\ y : \mathbf{1} \dashv$ $\Gamma, x : \texttt{chan}\ S$

Actris Rules (for HeapLang)

	Actris	**Session types**
Usage	$c \rightarrowtail \mathit{prot}$	$c\ :\ \texttt{chan}\ S$
New	$\{\mathrm{True}\}$ $\quad\texttt{new\_chan}\ ()$ $\{(c, c').\ c \rightarrowtail \mathit{prot} * c' \rightarrowtail \overline{\mathit{prot}}\}$	$\Gamma \vdash \texttt{new\_chan}\ () : \texttt{chan}\ S \times \texttt{chan}\ \overline{S} \dashv \Gamma$
Send	$\{c \rightarrowtail \mathbf{!}\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\ \mathit{prot} * P[\vec{t}/\vec{x}]\}$ $\quad\texttt{send}\ c\ (v[\vec{t}/\vec{x}])$ $\{c \rightarrowtail \mathit{prot}[\vec{t}/\vec{x}]\}$	$\Gamma, x : \texttt{chan}\ (\mathbf{!}A.\ S), y : A \vdash \texttt{send}\ x\ y : \mathbf{1} \dashv$ $\Gamma, x : \texttt{chan}\ S$
Recv	$\{c \rightarrowtail \mathbf{?}\vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\ \mathit{prot}\}$ $\quad\texttt{recv}\ c$ $\{w.\ \exists(\vec{y} : \vec{\tau}).\ (w = v[\vec{y}/\vec{x}]) *$ $\qquad P[\vec{y}/\vec{x}] * c \rightarrowtail \mathit{prot}[\vec{y}/\vec{x}]\}$	$\Gamma, x : \texttt{chan}\ (\mathbf{?}A.\ S) \vdash \texttt{recv}\ x : A \dashv$ $\Gamma, x : \texttt{chan}\ S$

Example program - via Actris rules

Example program:

$$\texttt{let}\,(c, c') := \texttt{new\_chan}\;()\;\texttt{in}$$
$$\texttt{fork}\;\{\texttt{let}\,x := \texttt{recv}\;c'\;\texttt{in}\,\texttt{send}\;c'\;(x+2)\}\,; \quad \text{// Service thread}$$
$$\texttt{send}\;c\;40;\;\texttt{recv}\;c \quad \text{// Client thread}$$

Example program - via Actris rules

Example program:

$$\texttt{let}\,(c, c') := \texttt{new\_chan}\ ()\ \texttt{in}$$
$$\texttt{fork}\ \{\texttt{let}\, x := \texttt{recv}\ c'\ \texttt{in}\, \texttt{send}\ c'\ (x+2)\}\,; \qquad \text{// Service thread}$$
$$\texttt{send}\ c\ 40;\ \texttt{recv}\ c \qquad \text{// Client thread}$$

Dependent separation protocols:

$$c \rightarrowtail \mathbf{!}\,(x:\mathbb{Z})\,\langle x\rangle\{\mathsf{True}\}.\,\mathbf{?}(y:\mathbb{Z})\,\langle y\rangle\{y=(x+2)\}.\,\mathbf{end} \qquad \texttt{and}$$
$$c' \rightarrowtail \mathbf{?}(x:\mathbb{Z})\,\langle x\rangle\{\mathsf{True}\}.\,\mathbf{!}\,(y:\mathbb{Z})\,\langle y\rangle\{y=(x+2)\}.\,\mathbf{end}$$

Example program - via Actris rules

Example program:

$$\texttt{let}\,(c, c') := \texttt{new\_chan}\ ()\ \texttt{in}$$
$$\texttt{fork}\ \{\texttt{let}\,x := \texttt{recv}\ c'\ \texttt{in}\,\texttt{send}\ c'\ (x+2)\}\,; \quad \text{// Service thread}$$
$$\texttt{send}\ c\ 40;\ \texttt{recv}\ c \quad \text{// Client thread}$$

Dependent separation protocols:

$$c \rightarrowtail \mathbf{!}\,(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,\mathbf{?}(y:\mathbb{Z})\,\langle y\rangle\{y=(x+2)\}.\,\mathbf{end} \qquad \texttt{and}$$
$$c' \rightarrowtail \mathbf{?}(x:\mathbb{Z})\,\langle x\rangle\{\mathrm{True}\}.\,\mathbf{!}\,(y:\mathbb{Z})\,\langle y\rangle\{y=(x+2)\}.\,\mathbf{end}$$

Properties obtained:

- ☑ Program does not crash

Example program - via Actris rules

Example program:

$$
\begin{array}{ll}
\texttt{let}\,(c, c') := \texttt{new\_chan}\ ()\ \texttt{in} & \\
\texttt{fork}\ \{\texttt{let}\, x := \texttt{recv}\ c'\ \texttt{in}\, \texttt{send}\ c'\ (x+2)\}\,; & //\ \text{Service thread} \\
\texttt{send}\ c\ 40;\ \texttt{recv}\ c & //\ \text{Client thread}
\end{array}
$$

Dependent separation protocols:

$$
\begin{array}{ll}
c \rightarrowtail \mathbf{!}\,(x : \mathbb{Z})\,\langle x \rangle \{\text{True}\}.\,\mathbf{?}(y : \mathbb{Z})\,\langle y \rangle \{y = (x+2)\}.\,\mathbf{end} & \texttt{and} \\
c' \rightarrowtail \mathbf{?}(x : \mathbb{Z})\,\langle x \rangle \{\text{True}\}.\,\mathbf{!}\,(y : \mathbb{Z})\,\langle y \rangle \{y = (x+2)\}.\,\mathbf{end} &
\end{array}
$$

Properties obtained:

- ☑ Program does not crash
- ☑ Program is correct (returns 42)

1. Dependent separation protocols
2. Actris Rules
3. Actris Ghost Theory
4. Mechanisation of Actris

The Actris Ghost Theory - Logical fragments

The logical fragments must capture the state of the reliable communication

The Actris Ghost Theory - Logical fragments

The logical fragments must capture the state of the reliable communication:

- The individual states of the protocols: $prot_1$ and $prot_2$

The Actris Ghost Theory - Logical fragments

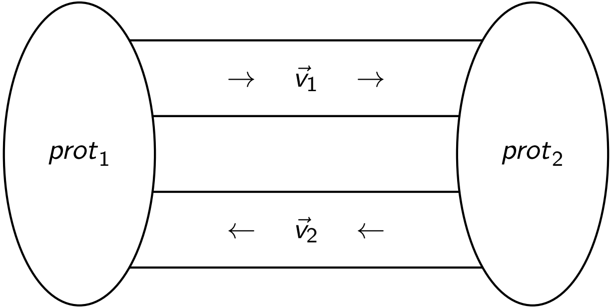
The logical fragments must capture the state of the reliable communication:

- The individual states of the protocols: $prot_1$ and $prot_2$
- The messages in transit in either direction: $\vec{v}_1$ and $\vec{v}_2$

The Actris Ghost Theory - Logical fragments

The logical fragments must capture the state of the reliable communication:

- The individual states of the protocols: $prot_1$ and $prot_2$
- The messages in transit in either direction: $\vec{v}_1$ and $\vec{v}_2$

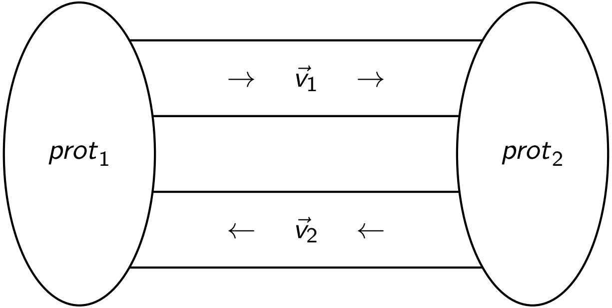


The logical fragments must capture the state of the reliable communication:

- The individual states of the protocols: $prot_1$ and $prot_2$
- The messages in transit in either direction: $\vec{v}_1$ and $\vec{v}_2$

Fragments:

$$t, u, P, Q ::= \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 \mid \mathsf{prot\_own}_\mathsf{l}\ \chi\ prot_1 \mid \mathsf{prot\_own}_\mathsf{r}\ \chi\ prot_2 \mid \ldots$$



The Actris Ghost Theory - Logical fragments

The logical fragments must capture the state of the reliable communication:

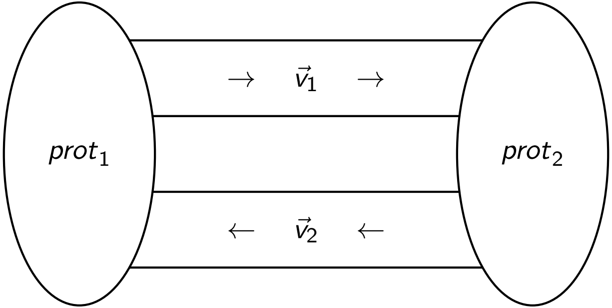
- The individual state

The ghost state identifier (χ) associates the fragments

- The messages in transit in either direction: $\vec{v}_1$ and $\vec{v}_2$

Fragments:

$$t, u, P, Q ::= \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 \mid \mathsf{prot\_own_l}\ \chi\ \mathit{prot}_1 \mid \mathsf{prot\_own_r}\ \chi\ \mathit{prot}_2 \mid \ldots$$



Fragments:

$$t, u, P, Q ::= \ \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 \mid \mathsf{prot\_own}_{\mathsf{l}}\ \chi\ \mathit{prot}_1 \mid \mathsf{prot\_own}_{\mathsf{r}}\ \chi\ \mathit{prot}_2 \mid \ldots$$

NB: only the rules of the left protocol are shown, as the right ones are symmetric

The Actris Ghost Theory - Rules

Fragments:

$$t, u, P, Q ::= \ \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 \mid \mathsf{prot\_own}_{\mathsf{l}}\ \chi\ \mathit{prot}_1 \mid \mathsf{prot\_own}_{\mathsf{r}}\ \chi\ \mathit{prot}_2 \mid \ldots$$

Rules:

$$\mathsf{True} \Rrightarrow \exists \chi.\ \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own}_{\mathsf{l}}\ \chi\ \mathit{prot} * \mathsf{prot\_own}_{\mathsf{r}}\ \chi\ \overline{\mathit{prot}} \qquad (\textsc{new})$$

NB: only the rules of the left protocol are shown, as the right ones are symmetric

The Actris Ghost Theory - Rules

Fragments:

$$t, u, P, Q ::= \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 \mid \mathsf{prot\_own_l}\ \chi\ \mathit{prot}_1 \mid \mathsf{prot\_own_r}\ \chi\ \mathit{prot}_2 \mid \ldots$$

Rules:

$$\mathsf{True} \Rrightarrow \exists \chi.\ \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathsf{prot\_own_r}\ \chi\ \overline{\mathit{prot}} \qquad (\textsc{new})$$

View shifts ($\Rrightarrow$) can be made in between any program step:

$$\textsc{Ht-vs}\quad \frac{P \Rrightarrow P' \qquad \{P'\}\ e\ \{w.\ Q'\} \qquad (\forall w.\ Q' \Rrightarrow Q)}{\{P\}\ e\ \{w.\ Q\}}$$

NB: only the rules of the left protocol are shown, as the right ones are symmetric

The Actris Ghost Theory - Rules

Fragments:

$$t, u, P, Q ::= \ \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 \mid \mathsf{prot\_own}_{\mathsf{l}}\ \chi\ \mathit{prot}_1 \mid \mathsf{prot\_own}_{\mathsf{r}}\ \chi\ \mathit{prot}_2 \mid \ldots$$

Rules:

$$\mathsf{True} \Rrightarrow \exists \chi.\ \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own}_{\mathsf{l}}\ \chi\ \mathit{prot} * \mathsf{prot\_own}_{\mathsf{r}}\ \chi\ \overline{\mathit{prot}} \qquad (\textsc{new})$$

$$\begin{aligned}&\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own}_{\mathsf{l}}\ \chi\ (\mathbf{!}\, \vec{x}\!:\!\vec{\tau}\ \langle v\rangle\{P\}.\ \mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow\\ &\qquad \triangleright^{|\vec{v}_2|}\big(\mathsf{prot\_ctx}\ \chi\ (\vec{v}_1 \cdot [v[\vec{t}/\vec{x}]])\ \vec{v}_2\big) * \mathsf{prot\_own}_{\mathsf{l}}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}])\end{aligned} \qquad (\textsc{send})$$

NB: only the rules of the left protocol are shown, as the right ones are symmetric

The Actris Ghost Theory - Rules

Fragments:

$$t, u, P, Q ::= \ \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 \mid \mathsf{prot\_own_l}\ \chi\ \mathit{prot}_1 \mid \mathsf{prot\_own_r}\ \chi\ \mathit{prot}_2 \mid \ldots$$

Rules:

$$\mathsf{True} \Rrightarrow \exists \chi.\ \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathsf{prot\_own_r}\ \chi\ \overline{\mathit{prot}} \qquad (\textsc{new})$$

$$\begin{aligned}&\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_l}\ \chi\ (\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow \\ &\qquad \triangleright^{|\vec{v}_2|}\left(\mathsf{prot\_ctx}\ \chi\ (\vec{v}_1 \cdot [v[\vec{t}/\vec{x}]])\ \vec{v}_2\right) * \mathsf{prot\_own_l}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}])\end{aligned} \qquad (\textsc{send})$$

$$\begin{aligned}&\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ ([w]\cdot \vec{v}_2) * \mathsf{prot\_own_l}\ \chi\ (\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) \Rrightarrow \\ &\qquad \triangleright \exists(\vec{y} : \vec{\tau}).\,(w = v[\vec{y}/\vec{x}]) * P[\vec{y}/\vec{x}] * \\ &\qquad\qquad \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_l}\ \chi\ (\mathit{prot}[\vec{y}/\vec{x}])\end{aligned} \qquad (\textsc{recv})$$

NB: only the rules of the left protocol are shown, as the right ones are symmetric

The Actris Ghost Theory - Rules

Fragments:

$$t, u, P, Q ::= \ \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 \mid \mathsf{prot\_own_l}\ \chi\ \mathit{prot}_1 \mid \mathsf{prot\_own_r}\ \chi\ \mathit{prot}_2 \mid \ldots$$

Rules:

$$\mathsf{True} \Rrightarrow \exists \chi.\ \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathsf{prot\_own_r}\ \chi\ \overline{\mathit{prot}} \qquad (\textsc{new})$$

$$\begin{aligned} &\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_l}\ \chi\ (\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow \\ &\qquad \triangleright^{|\vec{v}_2|}\big(\mathsf{prot\_ctx}\ \chi\ (\vec{v}_1 \cdot [v[\vec{t}/\vec{x}]])\ \vec{v}_2\big) * \mathsf{prot\_own_l}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}]) \end{aligned} \qquad (\textsc{send})$$

$$\begin{aligned} &\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ ([w] \cdot \vec{v}_2) * \mathsf{prot\_own_l}\ \chi\ (\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) \Rrightarrow \\ &\qquad \triangleright \exists(\vec{y} : \vec{\tau}).\,(w = v[\vec{y}/\vec{x}]) * P[\vec{y}/\vec{x}] * \\ &\qquad\qquad \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_l}\ \chi\ (\mathit{prot}[\vec{y}/\vec{x}]) \end{aligned} \qquad (\textsc{recv})$$

$$\mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathit{prot} \sqsubseteq \mathit{prot}' \twoheadrightarrow \mathsf{prot\_own_l}\ \chi\ \mathit{prot}' \qquad (\textsc{subprotocol})$$

NB: only the rules of the left protocol are shown, as the right ones are symmetric

The Actris Ghost Theory - Rules

Fragments:

$$t, u, P, Q ::= \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 \mid \mathsf{prot\_own_l}\ \chi\ \mathit{prot}_1 \mid \mathsf{prot\_own_r}\ \chi\ \mathit{prot}_2 \mid \ldots$$

Rules:

Subprotocol relation ($\sqsubseteq$) inspired by asynchronous session subtyping

$$\mathsf{True} \Rrightarrow \exists \chi.\ \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathsf{prot\_own_r}\ \chi\ \mathit{prot} \qquad (\textsc{new})$$

$$\begin{array}{l}\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_l}\ \chi\ (\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow \\ \quad \triangleright^{|\vec{v}_2|}\big(\mathsf{prot\_ctx}\ \chi\ (\vec{v}_1 \cdot [v[\vec{t}/\vec{x}]])\ \vec{v}_2\big) * \mathsf{prot\_own_l}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}])\end{array} \qquad (\textsc{send})$$

$$\begin{array}{l}\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ ([w] \cdot \vec{v}_2) * \mathsf{prot\_own_l}\ \chi\ (\mathbf{?}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) \Rrightarrow \\ \quad \triangleright \exists(\vec{y} : \vec{\tau}).\,(w = v[\vec{y}/\vec{x}]) * P[\vec{y}/\vec{x}] * \\ \qquad\qquad \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_l}\ \chi\ (\mathit{prot}[\vec{y}/\vec{x}])\end{array} \qquad (\textsc{recv})$$

$$\mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathit{prot} \sqsubseteq \mathit{prot}' \twoheadrightarrow \mathsf{prot\_own_l}\ \chi\ \mathit{prot}' \qquad (\textsc{subprotocol})$$

NB: only the rules of the left protocol are shown, as the right ones are symmetric

The Actris Ghost Theory - Rules

Fragments:

$$t, u, P, Q ::= \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 \mid \mathsf{prot\_own_l}\ \chi\ \mathit{prot}_1 \mid \mathsf{prot\_own_r}\ \chi\ \mathit{prot}_2 \mid \ldots$$

Rules:

$$\mathsf{True} \Rrightarrow \exists \chi.\ \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathsf{prot\_own_r}\ \chi\ \overline{\mathit{prot}} \qquad (\textsc{new})$$

$$\begin{aligned}&\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_l}\ \chi\ (!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow \\ &\quad \triangleright^{|\vec{v}_2|}\,(\mathsf{prot\_ctx}\ \chi\ (\vec{v}_1 \cdot [v[\vec{t}/\vec{x}]])\ \vec{v}_2) * \mathsf{prot\_own_l}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}])\end{aligned} \qquad (\textsc{send})$$

$$\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ ([w] \cdot \vec{v}_2) * \mathsf{prot\_own_l}\ \chi\ (?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) \Rrightarrow$$

A later per inbound message as a side-effect of the protocols being higher-order

- Recent change to Iris: each step can strip laters based on total steps taken

$$\textsc{Ht-step-lb-get}\quad \frac{\{P * \text{⧗}\ 0\}\ e\ \{w.\ Q\}}{\{P\}\ e\ \{w.\ Q\}} \qquad \textsc{Ht-step-lb-incr}\quad \frac{\{P\}\ e\ \{w.\ Q * \text{⧗}\ (n+1)\}}{\{P * \text{⧗}\ n\}\ e\ \{w.\ Q\}} \qquad \textsc{Ht-step-lb-skip}\quad \frac{P_1 \Rrightarrow \triangleright^n R \qquad \{P_2\}\ e\ \{w.\ Q * R\}}{\{P_1 * P_2 * \text{⧗}\ n\}\ e\ \{w.\ Q\}}$$

NB: only the rules of the left protocol are shown, as the right ones are symmetric

Fragments:

Lower bound of total steps taken ($\text{⧖}\, n$)

$$t, u, P, Q ::= \ \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \ \vec{v}_2 \mid \mathsf{prot\_own_l}\ \chi\ \mathit{prot}_1 \mid \mathsf{prot\_own_r}\ \chi\ \mathit{prot}_2 \mid \ldots$$

Rules:

$$\mathsf{True} \Rrightarrow \exists \chi.\ \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathsf{prot\_own_r}\ \chi\ \overline{\mathit{prot}} \qquad (\textsc{new})$$

$$\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \ \vec{v}_2 * \mathsf{prot\_own_l}\ \chi\ (!\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow$$
$$\triangleright^{|\vec{v}_2|}\, (\mathsf{prot\_ctx}\ \chi\ (\vec{v}_1 \cdot [v[\vec{t}/\vec{x}]])\ \vec{v}_2) * \mathsf{prot\_own_l}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}]) \qquad (\textsc{send})$$

$$\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ ([w] \cdot \vec{v}_2) * \mathsf{prot\_own_l}\ \chi\ (?\, \vec{x}{:}\vec{\tau}\, \langle v \rangle \{P\}.\, \mathit{prot}) \Rrightarrow$$

A later per inbound message as a side-effect of the protocols being higher-order

- Recent change to Iris: each step can strip laters based on total steps taken

Ht-step-lb-get

$$\frac{\{P * \text{⧖}\ 0\}\ e\ \{w.\ Q\}}{\{P\}\ e\ \{w.\ Q\}}$$

Ht-step-lb-incr

$$\frac{\{P\}\ e\ \{w.\ Q * \text{⧖}\ (n+1)\}}{\{P * \text{⧖}\, n\}\ e\ \{w.\ Q\}}$$

Ht-step-lb-skip

$$\frac{P_1 \Rrightarrow\ \triangleright^{n} R \qquad \{P_2\}\ e\ \{w.\ Q * R\}}{\{P_1 * P_2 * \text{⧖}\, n\}\ e\ \{w.\ Q\}}$$

NB: only the rules of the left protocol are shown, as the right ones are symmetric

The Actris Ghost Theory - Rules

Fragments:

Lower bound of total steps taken ($\text{⧗}\, n$)

$$t, u, P, Q ::= \ \ldots \mid \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 \mid \mathsf{prot\_own_l}\ \chi\ \mathit{prot}_1 \mid \mathsf{prot\_own_r}\ \chi\ \mathit{prot}_2 \mid \ldots$$

Rules:

Strip laters corresponding to lower bound

$$\mathsf{True} \Rrightarrow \exists \chi.\ \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathsf{prot\_own_r}\ \chi\ \overline{\mathit{prot}} \qquad (\textsc{new})$$

$$\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \mathsf{prot\_own_l}\ \chi\ (!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ \mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow$$
$$\triangleright^{|\vec{v}_2|}\ (\mathsf{prot\_ctx}\ \chi\ (\vec{v}_1 \cdot [v[\vec{t}/\vec{x}]])\ \vec{v}_2) * \mathsf{prot\_own_l}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}]) \qquad (\textsc{send})$$

$$\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ ([w] \cdot \vec{v}_2) * \mathsf{prot\_own_l}\ \chi\ (?\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\ \mathit{prot}) \Rrightarrow$$

A later per inbound message as a side-effect of the protocols being higher-order

- Recent change to Iris: each step can strip laters based on total steps taken

Ht-step-lb-get

$$\frac{\{P * \text{⧗}\ 0\}\ e\ \{w.\ Q\}}{\{P\}\ e\ \{w.\ Q\}}$$

Ht-step-lb-incr

$$\frac{\{P\}\ e\ \{w.\ Q * \text{⧗}\ (n+1)\}}{\{P * \text{⧗}\, n\}\ e\ \{w.\ Q\}}$$

Ht-step-lb-skip

$$\frac{P_1 \Rrightarrow \triangleright^n R \qquad \{P_2\}\ e\ \{w.\ Q * R\}}{\{P_1 * P_2 * \text{⧗}\, n\}\ e\ \{w.\ Q\}}$$

NB: only the rules of the left protocol are shown, as the right ones are symmetric

Proving the **Actris Rules** for shared memory message passing in HeapLang

Shared memory message passing in HeapLang

We must first provide an implementation of the message passing primitives

```
new_chan ()
```

```
send c v
```

```
recv c
```

Shared memory message passing in HeapLang

We must first provide an implementation of the message passing primitives

$$\begin{aligned}\texttt{new\_chan}\ () :=\ & \textcolor{blue}{\texttt{let}}\ (l, r, lk) := (\texttt{lnil}\ (), \texttt{lnil}\ (), \texttt{new\_lock}\ ())\ \textcolor{blue}{\texttt{in}}\\ & ((l, r, lk), (r, l, lk))\end{aligned}$$

$\texttt{send}\ c\ v$

$\texttt{recv}\ c$

Shared memory message passing in HeapLang

We must first provide an implementation of the message passing primitives

$$
\begin{aligned}
\texttt{new\_chan}\ () &:= \texttt{let}\ (l, r, lk) := (\texttt{lnil}\ (), \texttt{lnil}\ (), \texttt{new\_lock}\ ())\ \texttt{in} \\
&\ ((l, r, lk), (r, l, lk)) \\
\texttt{send}\ c\ v &:= \texttt{let}\ (l, r, lk) := c\ \texttt{in} \\
&\ \texttt{acquire}\ lk; \\
&\ \quad \texttt{lsnoc}\ l\ v; \\
&\ \texttt{release}\ lk \\
\texttt{recv}\ c &
\end{aligned}
$$

Shared memory message passing in HeapLang

We must first provide an implementation of the message passing primitives

$$
\begin{aligned}
\texttt{new\_chan}\;() &:= \texttt{let}\,(l, r, lk) := (\texttt{lnil}\;(), \texttt{lnil}\;(), \texttt{new\_lock}\;())\;\texttt{in}\\
&\quad ((l, r, lk), (r, l, lk))\\
\texttt{send}\;c\;v &:= \texttt{let}\,(l, r, lk) := c\;\texttt{in}\\
&\quad \texttt{acquire}\;lk;\\
&\quad\quad \texttt{lsnoc}\;l\;v;\\
&\quad \texttt{release}\;lk\\
\texttt{recv}\;c &:= \texttt{match}\,(\texttt{try\_recv}\;c)\,\texttt{with}\\
&\quad\ \ \texttt{inj}_1\,() \Rightarrow \texttt{recv}\;c\\
&\quad \mid \texttt{inj}_2\,v \ \Rightarrow v\\
&\quad \texttt{end}\\
\texttt{try\_recv}\;c &:= \texttt{let}\,(l, r, lk) := c\;\texttt{in}\\
&\quad \texttt{acquire}\;lk;\\
&\quad\quad \texttt{let}\;ret := (\texttt{if}\,(\texttt{lisnil}\;r)\,\texttt{then}\,(\texttt{inj}_1\,())\,\texttt{else}\,(\texttt{inj}_2\,(\texttt{lpop}\;r)))\;\texttt{in}\\
&\quad \texttt{release}\;lk;\;\; ret
\end{aligned}
$$

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathsf{prot\_own}_{\mathsf{l}}\ \chi\ \mathit{prot}$ / $\mathsf{prot\_own}_{\mathsf{r}}\ \chi\ \mathit{prot}$

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathsf{prot\_own_l}\ \chi\ \mathit{prot}$ / $\mathsf{prot\_own_r}\ \chi\ \mathit{prot}$
  - Include the shared protocol context: $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathsf{prot\_own_l}\ \chi\ \mathit{prot}$ / $\mathsf{prot\_own_r}\ \chi\ \mathit{prot}$
  - Include the shared protocol context: $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$
  - Include the step lower bound for each logical buffer: $⧖\,|\vec{v}_1|$ and $⧖\,|\vec{v}_2|$

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathsf{prot\_own_l}\ \chi\ \mathit{prot}$ / $\mathsf{prot\_own_r}\ \chi\ \mathit{prot}$
  - Include the shared protocol context: $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$
  - Include the step lower bound for each logical buffer: ⧖$|\vec{v}_1|$ and ⧖$|\vec{v}_2|$
- **Implementation-specific physical state**
  - Capture the structure of the channel abstraction c

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathsf{prot\_own_l}\ \chi\ \mathit{prot}$ / $\mathsf{prot\_own_r}\ \chi\ \mathit{prot}$
  - Include the shared protocol context: $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$
  - Include the step lower bound for each logical buffer: $⧖|\vec{v}_1|$ and $⧖|\vec{v}_2|$
- **Implementation-specific physical state**
  - Capture the structure of the channel abstraction c
  - Connect the physical state to the logical buffers

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathsf{prot\_own_l}\ \chi\ \mathit{prot}$ / $\mathsf{prot\_own_r}\ \chi\ \mathit{prot}$
  - Include the shared protocol context: $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$
  - Include the step lower bound for each logical buffer: $⧖\,|\vec{v}_1|$ and $⧖\,|\vec{v}_2|$
- **Implementation-specific physical state**
  - Capture the structure of the channel abstraction c
  - Connect the physical state to the logical buffers
  - Include a means of synchronisation between the two endpoints

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathsf{prot\_own}_\mathsf{l}\ \chi\ \mathit{prot}$ / $\mathsf{prot\_own}_\mathsf{r}\ \chi\ \mathit{prot}$
  - Include the shared protocol context: $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$
  - Include the step lower bound for each logical buffer: $⧖|\vec{v}_1|$ and $⧖|\vec{v}_2|$
- **Implementation-specific physical state (for HeapLang)**
  - Capture the structure of the channel abstraction c: (l, r, lk) / (r, l, lk)
  - Connect the physical state to the logical buffers
  - Include a means of synchronisation between the two endpoints

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathsf{prot\_own_l}\ \chi\ \mathit{prot}$ / $\mathsf{prot\_own_r}\ \chi\ \mathit{prot}$
  - Include the shared protocol context: $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$
  - Include the step lower bound for each logical buffer: $⧖\,|\vec{v}_1|$ and $⧖\,|\vec{v}_2|$
- **Implementation-specific physical state (for HeapLang)**
  - Capture the structure of the channel abstraction c: (l, r, lk) / (r, l, lk)
  - Connect the physical state to the logical buffers: $\texttt{isList}\ l\ \vec{v}_1$ / $\texttt{isList}\ r\ \vec{v}_2$
  - Include a means of synchronisation between the two endpoints

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathsf{prot\_own_l}\ \chi\ \mathit{prot}$ / $\mathsf{prot\_own_r}\ \chi\ \mathit{prot}$
  - Include the shared protocol context: $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$
  - Include the step lower bound for each logical buffer: $⧖\,|\vec{v}_1|$ and $⧖\,|\vec{v}_2|$
- **Implementation-specific physical state (for HeapLang)**
  - Capture the structure of the channel abstraction c: (l, r, lk) / (r, l, lk)
  - Connect the physical state to the logical buffers: $\mathtt{isList}\ l\ \vec{v}_1$ / $\mathtt{isList}\ r\ \vec{v}_2$
  - Include a means of synchronisation between the two endpoints

List ownership ($\mathtt{isList}\ l\ \vec{x}$) asserts exclusive ownership of the list l with contents $\vec{x}$

Ht-lnil
$\{\mathsf{True}\}\ \mathtt{lnil}\ \{l.\ \mathtt{isList}\ l\ []\}$

Ht-lsnoc
$\{\mathtt{isList}\ l\ \vec{x} * l \mapsto x\ v\}\ \mathtt{lsnoc}\ l\ v\ \{\mathtt{isList}\ l\ (\vec{x} \cdot [x])\}$

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathrm{prot\_own}_{\mathrm{l}}\ \chi\ \mathit{prot}$ / $\mathrm{prot\_own}_{\mathrm{r}}\ \chi\ \mathit{prot}$
  - Include the shared protocol context: $\mathrm{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$
  - Include the step lower bound for each logical buffer: $⧗\,|\vec{v}_1|$ and $⧗\,|\vec{v}_2|$
- **Implementation-specific physical state (for HeapLang)**
  - Capture the structure of the channel abstraction c: (l, r, lk) / (r, l, lk)
  - Connect the physical state to the logical buffers: $\texttt{isList}\ l\ \vec{v}_1$ / $\texttt{isList}\ r\ \vec{v}_2$
  - Include a means of synchronisation between the two endpoints: $\mathrm{is\_lock}\ lk\ R$

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathsf{prot\_own}_{\mathsf{l}}\ \chi\ \mathit{prot}$ / $\mathsf{prot\_own}_{\mathsf{r}}\ \chi\ \mathit{prot}$
  - Include the shared protocol context: $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$
  - Include the step lower bound for each logical buffer: $⧖\,|\vec{v}_1|$ and $⧖\,|\vec{v}_2|$
- **Implementation-specific physical state (for HeapLang)**
  - Capture the structure of the channel abstraction c: (l, r, lk) / (r, l, lk)
  - Connect the physical state to the logical buffers: $\texttt{isList}\ l\ \vec{v}_1$ / $\texttt{isList}\ r\ \vec{v}_2$
  - Include a means of synchronisation between the two endpoints: $\mathsf{is\_lock}\ lk\ R$

Lock ownership ($\mathsf{is\_lock}\ lk\ R$) asserts that the lock lk governs the proposition R

Ht-acquire
$\{\mathsf{is\_lock}\ lk\ R\}\ \texttt{acquire}\ lk\ \{R\}$

Ht-release
$\{\mathsf{is\_lock}\ lk\ R * R\}\ \texttt{release}\ lk\ \{\mathsf{True}\}$

Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\mathsf{prot\_own_l}\ \chi\ \mathit{prot}$ / $\mathsf{prot\_own_r}\ \chi\ \mathit{prot}$
  - Include the shared protocol context: $\mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2$
  - Include the step lower bound for each logical buffer: $⧖\,|\vec{v}_1|$ and $⧖\,|\vec{v}_2|$
- **Implementation-specific physical state (for HeapLang)**
  - Capture the structure of the channel abstraction c: (l, r, lk) / (r, l, lk)
  - Connect the physical state to the logical buffers: $\texttt{isList}\ l\ \vec{v}_1$ / $\texttt{isList}\ r\ \vec{v}_2$
  - Include a means of synchronisation between the two endpoints: $\mathsf{is\_lock}\ lk\ R$

In the case of the HeapLang implementation it can then be defined as follows:

$$
\begin{aligned}
c \rightarrowtail \mathit{prot} \triangleq \exists \chi, l, r, lk.\ & \begin{pmatrix} (c = (l, r, lk) * \mathsf{prot\_own_l}\ \chi\ \mathit{prot}) \vee \\ (c = (r, l, lk) * \mathsf{prot\_own_r}\ \chi\ \mathit{prot}) \end{pmatrix} * \\
& \mathsf{is\_lock}\ lk\ (\exists \vec{v}_1\ \vec{v}_2.\ \texttt{isList}\ l\ \vec{v}_1 * \texttt{isList}\ r\ \vec{v}_2 * \\
& \qquad\qquad \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * ⧖\,|\vec{v}_1| * ⧖\,|\vec{v}_2|)
\end{aligned}
$$

Proving the Actris rules - newchan

We wish to prove:

$$\{\mathsf{True}\}\,\texttt{new\_chan}\ ()\,\{w.\,\exists c_1, c_2.\, w = (c_1, c_2) * c_1 \rightarrowtail \mathit{prot} * c_2 \rightarrowtail \overline{\mathit{prot}}\}$$

It follows almost directly from the rule:

$$\mathsf{True} \Rrightarrow \exists \chi.\, \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own}_{\mathsf{l}}\ \chi\ \mathit{prot} * \mathsf{prot\_own}_{\mathsf{r}}\ \chi\ \overline{\mathit{prot}}$$

And the definition of the channel endpoint ownership:

$$\begin{aligned} c \rightarrowtail \mathit{prot} \triangleq \exists \chi, l, r, lk.\, & \begin{pmatrix} (c = (l, r, lk) * \mathsf{prot\_own}_{\mathsf{l}}\ \chi\ \mathit{prot}) \vee \\ (c = (r, l, lk) * \mathsf{prot\_own}_{\mathsf{r}}\ \chi\ \mathit{prot}) \end{pmatrix} * \\ & \mathsf{is\_lock}\ lk\ (\exists \vec{v}_1\ \vec{v}_2.\, \texttt{isList}\ l\ \vec{v}_1 * \texttt{isList}\ r\ \vec{v}_2\ * \\ & \qquad \mathsf{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * ⧖\,|\vec{v}_1| * ⧖\,|\vec{v}_2|) \end{aligned}$$

Proving the Actris rules - send

We wish to prove:

$$\left\{c \rightarrowtail \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} * P[\vec{t}/\vec{x}]\right\}\ \texttt{send}\ c\ (v[\vec{t}/\vec{x}])\ \left\{c \rightarrowtail \mathit{prot}[\vec{t}/\vec{x}]\right\}$$

It follows almost directly from the rule:

$$\begin{aligned}&\text{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * \text{prot\_own}_{\text{l}}\ \chi\ (\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow\\ &\quad \triangleright^{|\vec{v}_2|}\left(\text{prot\_ctx}\ \chi\ (\vec{v}_1 \cdot [v[\vec{t}/\vec{x}]])\ \vec{v}_2\right) * \text{prot\_own}_{\text{l}}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}])\end{aligned}$$

And the definition of the channel endpoint ownership:

$$\begin{aligned}c \rightarrowtail \mathit{prot} \triangleq \exists \chi, l, r, lk.\ &\begin{pmatrix}(c = (l, r, lk) * \text{prot\_own}_{\text{l}}\ \chi\ \mathit{prot}) \vee \\ (c = (r, l, lk) * \text{prot\_own}_{\text{r}}\ \chi\ \mathit{prot})\end{pmatrix} *\\ &\text{is\_lock}\ lk\ (\exists \vec{v}_1\,\vec{v}_2.\ \texttt{isList}\ l\ \vec{v}_1 * \texttt{isList}\ r\ \vec{v}_2 *\\ &\qquad\qquad \text{prot\_ctx}\ \chi\ \vec{v}_1\ \vec{v}_2 * ⧖\,|\vec{v}_1| * ⧖\,|\vec{v}_2|)\end{aligned}$$

We wish to prove:

$$\{c \rightarrowtail \mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}\}\;\texttt{recv}\;c\;\{w.\,\exists\vec{y}.\,w = v[\vec{y}/\vec{x}] * c \rightarrowtail \mathit{prot}[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\}$$

It follows almost directly from the rule:

$$\begin{aligned}&\mathsf{prot\_ctx}\;\chi\;\vec{v}_1\;([w]\cdot\vec{v}_2) * \mathsf{prot\_own}_{\mathsf{l}}\;\chi\;(\mathbf{?}\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) \Rrightarrow\\ &\quad\triangleright\exists(\vec{y}:\vec{\tau}).\,(w = v[\vec{y}/\vec{x}]) * P[\vec{y}/\vec{x}] *\\ &\qquad\qquad\mathsf{prot\_ctx}\;\chi\;\vec{v}_1\;\vec{v}_2 * \mathsf{prot\_own}_{\mathsf{l}}\;\chi\;(\mathit{prot}[\vec{y}/\vec{x}])\end{aligned}$$

And the definition of the channel endpoint ownership:

$$\begin{aligned}c \rightarrowtail \mathit{prot} \triangleq \exists\chi, l, r, lk.\,&\begin{pmatrix}(c = (l, r, lk) * \mathsf{prot\_own}_{\mathsf{l}}\;\chi\;\mathit{prot}) \vee\\ (c = (r, l, lk) * \mathsf{prot\_own}_{\mathsf{r}}\;\chi\;\mathit{prot})\end{pmatrix} *\\ &\mathsf{is\_lock}\;lk\;(\exists\vec{v}_1\;\vec{v}_2.\,\texttt{isList}\;l\;\vec{v}_1 * \texttt{isList}\;r\;\vec{v}_2 *\\ &\qquad\qquad\mathsf{prot\_ctx}\;\chi\;\vec{v}_1\;\vec{v}_2 * \boxtimes|\vec{v}_1| * \boxtimes|\vec{v}_2|)\end{aligned}$$

1. Dependent separation protocols
2. Actris Rules
3. Actris Ghost Theory
4. Mechanisation of Actris

Dependent separation protocols:

- Define the type of *prot* using Iris's recursive domain equation solver

Dependent separation protocols:

- Define the type of *prot* using Iris's recursive domain equation solver
- Define constructors, operations, and relations on *prot*
  - $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, $\overline{\mathit{prot}}$, and $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$

Mechanisation of Actris

Dependent separation protocols:

- Define the type of *prot* using Iris's recursive domain equation solver
- Define constructors, operations, and relations on *prot*
  - $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, $\overline{\mathit{prot}}$, and $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$

Actris Ghost Theory:

- Define a notion of *protocol consistency* via the subprotocol relation

Mechanisation of Actris

Dependent separation protocols:

- Define the type of *prot* using Iris's recursive domain equation solver
- Define constructors, operations, and relations on *prot*
  - $\mathbf{!}\, \vec{x}:\vec{\tau}\, \langle v \rangle \{P\}.\, prot$, $\overline{prot}$, and $prot_1 \sqsubseteq prot_2$

Actris Ghost Theory:

- Define a notion of *protocol consistency* via the subprotocol relation
- Define the fragments via protocol consistency and Iris's higher-order ghost state

Dependent separation protocols:

- Define the type of *prot* using Iris's recursive domain equation solver
- Define constructors, operations, and relations on *prot*
  - $\mathbf{!}\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, $\overline{\mathit{prot}}$, and $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$

Actris Ghost Theory:

- Define a notion of *protocol consistency* via the subprotocol relation
- Define the fragments via protocol consistency and Iris's higher-order ghost state
- Prove the ghost theory rules via properties of the protocol consistency

Mechanisation of Actris

Dependent separation protocols:

- Define the type of *prot* using Iris's recursive domain equation solver
- Define constructors, operations, and relations on *prot*
  - $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, $\overline{\mathit{prot}}$, and $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$

Actris Ghost Theory:

- Define a notion of *protocol consistency* via the subprotocol relation
- Define the fragments via protocol consistency and Iris's higher-order ghost state
- Prove the ghost theory rules via properties of the protocol consistency

Actris Rules (for HeapLang):

- Implement the communication primitives in HeapLang
  - e.g. `send` and `recv`

Mechanisation of Actris

Dependent separation protocols:

- Define the type of *prot* using Iris's recursive domain equation solver
- Define constructors, operations, and relations on *prot*
  - $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, $\overline{\mathit{prot}}$, and $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$

Actris Ghost Theory:

- Define a notion of *protocol consistency* via the subprotocol relation
- Define the fragments via protocol consistency and Iris's higher-order ghost state
- Prove the ghost theory rules via properties of the protocol consistency

Actris Rules (for HeapLang):

- Implement the communication primitives in HeapLang
  - e.g. `send` and `recv`
- Define the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ using the Actris ghost theory

Mechanisation of Actris

Dependent separation protocols:

- Define the type of *prot* using Iris's recursive domain equation solver
- Define constructors, operations, and relations on *prot*
  - $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$, $\overline{\mathit{prot}}$, and $\mathit{prot}_1 \sqsubseteq \mathit{prot}_2$

Actris Ghost Theory:

- Define a notion of *protocol consistency* via the subprotocol relation
- Define the fragments via protocol consistency and Iris's higher-order ghost state
- Prove the ghost theory rules via properties of the protocol consistency

Actris Rules (for your language!):

- Implement the communication primitives in your language!
  - e.g. `send` and `recv`
- Define the channel endpoint ownership $c \rightarrowtail \mathit{prot}$ using the Actris ghost theory
- Prove the Actris rules as lemmas in Iris, using the ghost theory rules

Publications

Actris: Session-Type Based Reasoning in Separation Logic

- ACM SIGPLAN Symposium on Principles of Programming Languages 2020 [POPL'20]

Machine-Checked Semantic Session Typing

- Certified Programs and Proofs Conference 2021 [CPP'21] (Distinguished paper award)

Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic

- Journal of Logical Methods in Computer Science [LMCS'22] (Pending copy-editing)

Actris: Session-Type Based Reasoning in Separation Logic

JONAS KASTBERG HINRICHSEN, IT University of Copenhagen, Denmark
JESPER BENGTSON, IT University of Copenhagen, Denmark
ROBBERT KREBBERS, Delft University of Technology, The Netherlands

Message passing is a useful abstraction to implement concurrent programs. For real-world systems, however, it is often combined with other programming and concurrency paradigms, such as higher-order functions, mutable state, shared-memory concurrency, and locks. We present **Actris**: a logic for proving functional correctness of programs that use a combination of the aforementioned features. Actris combines the power of modern concurrent separation logics with a first-class protocol mechanism—based on session types—for reasoning about message passing in the presence of other concurrency paradigms. We show that Actris provides a suitable level of abstraction by proving functional correctness of a variety of examples, including a distributed merge sort, a distributed load-balancing mapper, and a variant of the map-reduce model, using relatively simple specifications. Soundness of Actris is proved using a model of its protocol mechanism in the Iris framework. We mechanised the theory of Actris, together with tactics for symbolic execution of programs, as well as all examples in the paper, in the Coq proof assistant.

CCS Concepts: • **Theory of computation → Separation logic**; *Program verification*; Programming logic.

Additional Key Words and Phrases: Message passing, actor model, concurrency, session types, Iris

ACM Reference Format:
Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-Type Based Reasoning in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 6 (January 2020), 30 pages. https://doi.org/10.1145/3371074

1 INTRODUCTION

Message-passing programs are ubiquitous in modern computer systems, emphasising the impor-

ACTRIS 2.0: ASYNCHRONOUS SESSION-TYPE BASED REASONING IN SEPARATION LOGIC

JONAS KASTBERG HINRICHSEN, JESPER BENGTSON, AND ROBBERT KREBBERS

IT University of Copenhagen, Denmark
e-mail address: jkas@itu.dk

IT University of Copenhagen, Denmark
e-mail address: bengtson@itu.dk

Radboud University and Delft University of Technology, The Netherlands
e-mail address: mail@robbertkrebbers.nl

Abstract. Message passing is a useful abstraction for implementing concurrent programs. For real-world systems, however, it is often combined with other programming and concurrency paradigms, such as higher-order functions, mutable state, shared-memory concurrency, and locks. We present **Actris**: a logic for proving functional correctness of programs that use a combination of the aforementioned features. Actris combines the power of modern concurrent separation logics with a first-class protocol mechanism—based on session types—for reasoning about message passing in the presence of other concurrency paradigms. We show that Actris provides a suitable level of abstraction by proving functional correctness of a variety of examples, including a distributed merge sort, a distributed load-balancing mapper, and a variant of the map-reduce model, using concise specifications.

While Actris was already presented in a conference paper (POPL'20), this paper expands the prior presentation significantly. Moreover, it extends Actris to **Actris 2.0** with a notion of *subprotocols*—based on session-type subtyping—that permits additional flexibility when composing channel endpoints, and that takes full advantage of the asynchronous semantics of message passing in Actris. Soundness of Actris 2.0 is proved using a model of its protocol mechanism in the Iris framework. We have mechanised the theory of Actris, together with custom tactics, as well as all examples in the paper, in the Coq proof assistant.

Machine-Checked Semantic Session Typing

Jonas Kastberg Hinrichsen
IT University of Copenhagen
Denmark

Daniël Louwrink
University of Amsterdam
The Netherlands

Robbert Krebbers
Radboud University and Delft University of Technology
The Netherlands

Jesper Bengtson
IT University of Copenhagen
Denmark

Abstract

Session types—a family of type systems for message-passing concurrency—have been subject to many extensions, where each extension comes with a separate proof of type safety. These extensions cannot be readily combined, and their proofs of type safety are generally not machine checked, making their correctness less trustworthy. We overcome these shortcomings with a semantic approach to binary asynchronous affine session types, by developing a logical relations model in Coq using the Iris program logic. We demonstrate the power of our approach by combining various forms of polymorphism and recursion, asynchronous subtyping, references, and locks/mutexes. As an additional benefit of the semantic approach, we demonstrate how to manually prove typing judgements of racy, but safe, programs that cannot be type checked using only the rules of the type system.

CCS Concepts: • **Theory of computation → Separation logic**; *Program verification*; Programming logic.

Keywords: Message passing, concurrency, session types, separation logic, semantic typing, Iris, Coq

ACM Reference Format:
Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021. Machine-Checked Semantic Session

we believe the following challenges have not received the attention that they deserve:

1. There are many extensions of session types with *e.g.*, polymorphism [18], asynchronous subtyping [37], and sharing via locks [5]. While type safety has been proven for each extension in isolation, existing proofs cannot be readily composed with each other, nor with other substructural type systems like Affe, Alms, Linear Haskell, Plaid, Rust, Mezzo, Quill, or System F°.
2. Session types use substructural types to enforce a strict discipline of channel ownership. While conventional session-type systems can type check many functions, they inherently exclude some functions that do not obey the ownership discipline, even if they are safe.
3. Only few session-type systems and their safety proofs have been machine checked by a proof assistant, making their correctness less trustworthy.

We address these challenges by eschewing the traditional *syntactic approach* to type safety (using *progress and preservation*) and instead embrace the *semantic approach* to type safety [1–3], using *logical relations* defined in terms of a program logic [4, 14, 15].

The semantic approach addresses the challenges above as (1) typing judgements are definitions in the program logic,

Future work

The **Actris** story is not over

RefinedC-style proof automation for reliable communication

- Symbolically verified programs for a subset of the protocol specifications

Multi-party dependent separation protocols

- Communication protocols that describe more than two parties

Deadlock and resource-leak-freedom guarantees

- Guarantees that the communication is deadlock free
- Guarantees that terminated communication leaves no leftover resources

Formal generalisation of the channel primitives and ownership

- Parametric abstractions that scales to different languages

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