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)

```plaintext
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)

```heaplang
new-chan(), send c v, recv c
```

Example Program:

```heaplang
let (c, c') := new-chan () in
fork {let x := recv c' in send c' (x + 2)}; // Service thread
send c 40; recv c // Client thread
```
Session types

Syntax

\[ A ::= Z | B | 1 | \]

\[ \text{chan } S | \ldots \]
Session types

Syntax

\[ A ::= Z \mid B \mid 1 \mid \text{chan } S \mid \ldots \]

\[ S ::= !A. S \mid ?A. S \mid \text{end } \mid \ldots \]
Session types

Syntax

\[ A ::= \mathbf{Z} \mid \mathbf{B} \mid 1 \mid \]  
\[ \text{chan } S \mid \ldots \]  
\[ S ::= !A. S \mid \]  
\[ ?A. S \mid \]  
\[ \text{end} \mid \ldots \]  

Example

\[ \text{chan } (!\mathbf{Z}. ?\mathbf{Z}. \text{end}) \]
Session types

Syntax

\[ A ::= \textbf{Z} \mid \textbf{B} \mid \textbf{1} \mid \text{chan } S \mid \ldots \]

\[ S ::= !A. S \mid ?A. S \mid \text{end} \mid \ldots \]

Example

\[ \text{chan } (!Z. ?Z. \text{end}) \]

Usage

\[ c : \text{chan } S \]
Session types

Syntax

\[
A ::= \text{Z} \mid \text{B} \mid \text{1} \mid \\
\text{chan } S \mid \ldots
\]

\[
S ::= !A. S \mid \\
?A. S \mid \\
\text{end} \mid \ldots
\]

Duality

\[
!A. S = ?A. \overline{S} \]

\[
?A. S = !A. \overline{S} \]

\[
\text{end} = \text{end}
\]

Example

\[
\text{chan } (!Z. ?Z. \text{end})
\]

Usage

\[
c : \text{chan } S
\]
Session types

Syntax

\[ A ::= Z \mid B \mid 1 \mid \text{chan } S \mid \ldots \]

\[ S ::= !A. S \mid ?A. S \mid \text{end} \mid \ldots \]

Duality

\[ \overline{!A. S} = ?A. \overline{S} \]

\[ \overline{?A. S} = !A. \overline{S} \]

\[ \overline{\text{end}} = \text{end} \]

Rules (for shared memory message passing)

\[ \Gamma \vdash \text{new-chan } () : \text{chan } S \times \text{chan } \overline{S} \vdash \Gamma \]

Example

\[ \text{chan } (!Z. \ ?Z. \ \text{end}) \]

Usage

\[ c : \text{chan } S \]
Session types

Syntax

\[ A ::= Z \mid B \mid 1 \mid \text{channel } S \mid \ldots \]

\[ S ::= !A. S \mid \text{query } A. S \mid \text{end} \mid \ldots \]

Example

\[ \text{channel } (!Z. ?Z. \text{end}) \]

Usage

\[ c : \text{channel } S \]

Duality

\[ !A. S = ?A. \overline{S} \]
\[ ?A. S = !A. \overline{S} \]
\[ \text{end} = \text{end} \]

Rules (for shared memory message passing)

\[ \Gamma \vdash \text{new} \_\text{channel } () : \text{channel } S \times \text{channel } \overline{S} \vdash \Gamma \]
\[ \Gamma, x : \text{channel } (!A. S), y : A \vdash \text{send } x \_y : 1 \vdash \Gamma, x : \text{channel } S \]
Session types

Syntax

\[ A ::= \textbf{Z} \mid \textbf{B} \mid \mathbf{1} \mid \text{chan } S \mid \ldots \]

\[ S ::= \textbf{!}A.S \mid \textbf{?}A.S \mid \text{end} \mid \ldots \]

Example

\[ \text{chan } (\textbf{!Z. ?Z. end}) \]

Usage

\[ c : \text{chan } S \]

Duality

\[ \textbf{!}A.S = \textbf{?}A.\overline{S} \]

\[ \textbf{?}A.S = \textbf{!}A.\overline{S} \]

\[ \text{end} = \text{end} \]

Rules (for shared memory message passing)

\[ \Gamma \vdash \text{new-chan } () : \text{chan } S \times \text{chan } \overline{S} \vdash \Gamma \]

\[ \Gamma, x : \text{chan } (\textbf{!}A.S), y : A \vdash \text{send } x y : \mathbf{1} \vdash \Gamma, x : \text{chan } S \]

\[ \Gamma, x : \text{chan } (\textbf{?}A.S) \vdash \text{recv } x : A \vdash \Gamma, x : \text{chan } S \]
Session types

Syntax

\[ A ::= Z \mid B \mid 1 \mid \text{channel } S \mid \ldots \]

\[ S ::= !A. S \mid ?A. S \mid \text{end} \mid \ldots \]

Duality

\[ \overline{!A. S} = ?A. \overline{S} \]
\[ \overline{?A. S} = !A. \overline{S} \]
\[ \text{end} = \text{end} \]

Rules (for shared memory message passing)

\[ \Gamma \vdash \text{new channel } () : \text{channel } S \times \text{channel } \overline{S} \vdash \Gamma \]
\[ \Gamma, x : \text{channel } (!A. S), y : A \vdash \text{send } x \ y : 1 \vdash \Gamma, x : \text{channel } S \]
\[ \Gamma, x : \text{channel } (?A. S) \vdash \text{recv } x : A \vdash \Gamma, x : \text{channel } S \]

Example program (service thread)

\[ \lambda c. \text{let } x := \text{recv } c \text{ in } \]
\[ \text{send } c \ (x + 2) \]
Session types

Syntax

\[ A ::= Z | B | 1 | \]
\[ \text{chan } S | \ldots \]
\[ S ::= !A. S \quad | \quad ?A. S \quad | \quad \text{end} \quad | \ldots \]

Example

\[ \text{chan } (!Z. ?Z. \text{end}) \]

Usage

\[ c : \text{chan } S \]

Duality

\[ !A. S = ?A. \overline{S} \]
\[ ?A. S = !A. \overline{S} \]
\[ \text{end} = \text{end} \]

Rules (for shared memory message passing)

\[ \Gamma \vdash \text{new channel } () : \text{chan } S \times \text{chan } \overline{S} \vdash \Gamma \]
\[ \Gamma, x : \text{chan } (!A. S), y : A \vdash \text{send } x \, y : 1 \vdash \Gamma, x : \text{chan } S \]
\[ \Gamma, x : \text{chan } (?A. S) \vdash \text{recv } x : A \vdash \Gamma, x : \text{chan } S \]

Example program (service thread)

\[ \Gamma \vdash \lambda c. \text{let } x := \text{recv } c \text{ in } \]
\[ \quad \text{send } c \, (x + 2) : \text{chan } (?Z. !Z. \text{end}) \rightarrow 1 \vdash \Gamma \]
Example program - via session types

Example program:

```ml
let (c, c') := new_chan () in
fork {let x := recv c' in send c' (x + 2)}; // Service thread
send c 40; recv c // Client thread
```
Example program - via session types

Example program:

```plaintext
let (c, c') := new_chan () in
fork {let x := recv c' in send c' (x + 2)};  // Service thread
send c 40; recv c                       // Client thread
```

Session types:

```
c : chan (!Z.?Z.end) and
```
```
c' : chan (?Z.!Z.end)
```
Example program - via session types

Example program:

```ml
let (c, c') := new_chan () in
fork {let x := recv c' in send c' (x + 2)};  // Service thread
send c 40; recv c                           // Client thread
```

Session types:

- \( c : \text{chan} (\text{!Z. ?Z. end}) \) and
- \( c' : \text{chan} (\text{?Z. !Z. end}) \)

Properties obtained:

- ✓ Program does not crash
Example program - via session types

Example program:

```plaintext
let (c, c') := new_chan () in
fork {let x := recv c' in send c' (x + 2)};  // Service thread
send c 40; recv c  // Client thread
```

Session types:

- $c : \text{chan (}!Z. ?Z. \text{end})$ and
- $c' : \text{chan (}?Z. !Z. \text{end})$

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

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

**Session types**
- Modular verification of channel endpoints
- Ensures safety
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/](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/](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 ($\vec{x} : \vec{\tau}$)

Syntax

$\text{prot} \ ::= \ ! \vec{x} : \vec{\tau} \langle \vec{v} \rangle \{ P \} . \text{prot} | \ ? \vec{x} : \vec{\tau} \langle \vec{v} \rangle \{ P \} . \text{prot} | \text{end}$

Example

$\! (x : \mathbb{Z}) \langle x \rangle \{ \text{True} \} . \ ? (y : \mathbb{Z}) \langle y \rangle \{ y = (x + 2) \} . \text{end}$

$\! \mathbb{Z}. \ ? \mathbb{Z}. \text{end}$

Duality

$\vec{x} : \vec{\tau} \langle \vec{v} \rangle \{ P \} . \text{prot} = \ ? \vec{x} : \vec{\tau} \langle \vec{v} \rangle \{ P \} . \text{prot}$

$\ ? \vec{x} : \vec{\tau} \langle \vec{v} \rangle \{ P \} . \text{prot} = \ ! \vec{x} : \vec{\tau} \langle \vec{v} \rangle \{ P \} . \text{prot}$

$\text{end} = \text{end}$

$\! A. S = \ ? A. S \ ? A. S = \ ! A. S \end{eqnarray}$
Dependent separation protocols

Session type-inspired protocols for functional correctness:

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Dependent separation protocols</th>
<th>Session types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$prot ::= !\vec{x}:\vec{\tau}\langle v \rangle{P}.\ prot$</td>
<td>$S ::= !A.\ S$</td>
<td>$?A.\ S$</td>
</tr>
<tr>
<td>$?\vec{x}:\vec{\tau}\langle v \rangle{P}.\ prot$</td>
<td>$\quad$</td>
<td>$\quad$</td>
</tr>
<tr>
<td>end</td>
<td>$\quad$</td>
<td>$end$</td>
</tr>
</tbody>
</table>
Dependent separation protocols

Session type-inspired protocols for functional correctness:

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Dependent separation protocols</th>
<th>Session types</th>
</tr>
</thead>
<tbody>
<tr>
<td>*prot ::= ! $\vec{x} : \vec{\tau} \langle v \rangle { P }$. prot</td>
<td>$S ::= ! A. S$</td>
<td>![Z]. ?Z. end</td>
</tr>
<tr>
<td></td>
<td>?$\vec{x} : \vec{\tau} \langle v \rangle { P }$. prot</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td><img src="x" alt="Z" /> ⟨x⟩{True}. ?(y:Z) ⟨y⟩{y = (x + 2)}. end</td>
<td></td>
</tr>
</tbody>
</table>
Dependent separation protocols

Session type-inspired protocols for functional correctness:

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

**Syntax**

\[
\text{prot ::= } !\vec{x}:\vec{\tau}\langle v\rangle\{P\}.\text{prot} \quad \mid \quad \text{end}
\]

**Example**

\[
! (x: \mathbb{Z})\langle x\rangle\{\text{True}\}. ?(y: \mathbb{Z})\langle y\rangle\{y = (x + 2)\}.\text{end}
\]

**Duality**

\[
\begin{align*}
!\vec{x}:\vec{\tau}\langle v\rangle\{P\}.\text{prot} & = \ ?\vec{x}:\vec{\tau}\langle v\rangle\{P\}.\overline{\text{prot}} \\
?\vec{x}:\vec{\tau}\langle v\rangle\{P\}.\text{prot} & = \ !\vec{x}:\vec{\tau}\langle v\rangle\{P\}.\overline{\text{prot}} \\
\text{end} & = \text{end}
\end{align*}
\]

**Session types**

\[
\begin{align*}
S ::= & \ !A. S \quad \mid \quad \text{end} \\
?A. S \quad \mid \quad \text{end} & = \ldots
\end{align*}
\]

\[
\begin{align*}
!A. S & = ?A. \overline{S} \\
?A. S & = !A. \overline{S} \\
\text{end} & = \text{end}
\end{align*}
\]
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 $prot$

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$prot ::= ! \vec{x}:\vec{\tau} \langle v \rangle { P } . \overline{prot}$</td>
</tr>
<tr>
<td>$?\vec{x}:\vec{\tau} \langle v \rangle { P } . \overline{prot}$</td>
</tr>
<tr>
<td>$\text{end}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$! (x: \mathbb{Z}) \langle x \rangle { \text{True} } . ?(y: \mathbb{Z}) \langle y \rangle { y = (x + 2) }. \text{end}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Duality</th>
</tr>
</thead>
<tbody>
<tr>
<td>$! \vec{x}:\vec{\tau} \langle v \rangle { P } . \overline{prot} = ?\vec{x}:\vec{\tau} \langle v \rangle { P } . \overline{prot}$</td>
</tr>
<tr>
<td>$?\vec{x}:\vec{\tau} \langle v \rangle { P } . \overline{prot} = ! \vec{x}:\vec{\tau} \langle v \rangle { P } . \overline{prot}$</td>
</tr>
<tr>
<td>$\text{end} = \text{end}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Session types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S ::= ! A. S$</td>
</tr>
<tr>
<td>$?A. S$</td>
</tr>
<tr>
<td>$\text{end} \ldots$</td>
</tr>
</tbody>
</table>

| $!Z. ?Z. \text{end}$ |
| $!A. S = ?A. \overline{S}$ |
| $?A. \overline{S} = !A. S$ |
| $\text{end} = \text{end}$ |
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

### Syntax

<table>
<thead>
<tr>
<th>Protocols</th>
<th>Session types</th>
</tr>
</thead>
</table>
| ```markdown
Dependent separation protocols
```
| ```markdown
Syntax
```
| prot ::= ! $\vec{x}:\vec{\tau}\langle v \rangle\{P\}$. prot |
|        | ?$\vec{x}:\vec{\tau}\langle v \rangle\{P\}$. prot |
|        | end |
| ```markdown
Example
```
| ```markdown
Example
```
| !$(x:\mathbb{Z})\langle x \rangle\{\text{True}\}$. ?$(y:\mathbb{Z})\langle y \rangle\{y = (x + 2)\}$. end |
| ```markdown
Duality
```
| ```markdown
Duality
```
| !$\vec{x}:\vec{\tau}\langle v \rangle\{P\}$. prot | = | ?$\vec{x}:\vec{\tau}\langle v \rangle\{P\}$. prot |
| ?$\vec{x}:\vec{\tau}\langle v \rangle\{P\}$. prot | = | !$\vec{x}:\vec{\tau}\langle v \rangle\{P\}$. prot |
| end | = | end |

```markdown
Session types
```
| ```markdown
Session types
```
| S ::= !A. S |
| ?A. S |
| end |
| ... |
1. Dependent separation protocols
2. Actris Rules
3. Actris Ghost Theory
4. Mechanisation of Actris
## Actris Rules (for HeapLang)

<table>
<thead>
<tr>
<th><strong>Usage</strong></th>
<th><strong>Actris</strong></th>
<th><strong>Session types</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$c \mapsto \text{prot}$</td>
<td>$c : \text{chan } S$</td>
</tr>
</tbody>
</table>
# Actris Rules (for HeapLang)

<table>
<thead>
<tr>
<th>Actris</th>
<th>Session types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usage</strong></td>
<td></td>
</tr>
<tr>
<td>$c \mapsto prot$</td>
<td>$c : \text{chan } S$</td>
</tr>
<tr>
<td>${\text{True}}$</td>
<td></td>
</tr>
<tr>
<td><strong>New</strong></td>
<td></td>
</tr>
<tr>
<td>$\text{new_chan } ()$</td>
<td>$\Gamma \vdash \text{new_chan } () : \text{chan } S \times \text{chan } \overline{S} \to \Gamma$</td>
</tr>
<tr>
<td>${(c, c'). c \mapsto prot * c' \mapsto \overline{prot}}$</td>
<td></td>
</tr>
<tr>
<td><strong>Actris</strong></td>
<td><strong>Session types</strong></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td><strong>Usage</strong></td>
<td></td>
</tr>
<tr>
<td>$c \mapsto prot$</td>
<td>$c : \text{chan } S$</td>
</tr>
<tr>
<td>${ \text{True} }$</td>
<td></td>
</tr>
<tr>
<td><strong>New</strong></td>
<td>$\Gamma \vdash \text{new_chan }() : \text{chan } S \times \text{chan } \overline{S} \vdash \Gamma$</td>
</tr>
<tr>
<td>${(c, c'). c \mapsto prot * c' \mapsto \overline{prot} }$</td>
<td></td>
</tr>
<tr>
<td><strong>Send</strong></td>
<td></td>
</tr>
<tr>
<td>${ c \mapsto ! \overline{x} : \overline{\tau} \langle v \rangle { P }. \text{prot} * P[\overline{t}/\overline{x}] }$</td>
<td>$\Gamma, x : \text{chan } (!A. S), y : A \vdash \text{send } x \ y : 1 \vdash$</td>
</tr>
<tr>
<td>$\text{send } c (v[\overline{t}/\overline{x}])$</td>
<td>$\Gamma, x : \text{chan } S$</td>
</tr>
<tr>
<td>${ c \mapsto \text{prot}[\overline{t}/\overline{x}] }$</td>
<td></td>
</tr>
</tbody>
</table>
# Actris Rules (for HeapLang)

**Usage**

\[ c \mapsto \text{prot} \]

\[ \{ \text{True} \} \]

**New**

\[ \text{new\_chan}() \]

\[ \{(c, c'). c \mapsto \text{prot} * c' \mapsto \overline{\text{prot}}\} \]

\[ \{ c \mapsto !\overline{x} : \overline{\tau}(v)\{P\}. \text{prot} * P[\overline{t}/\overline{x}]\} \]

\[ \text{send}\ c\ (v[\overline{t}/\overline{x}]) \]

\[ \{ c \mapsto \text{prot}[\overline{t}/\overline{x}]\} \]

**Send**

\[ \{ c \mapsto ?\overline{x} : \overline{\tau}(v)\{P\}. \text{prot}\} \]

\[ \text{recv}\ c \]

\[ \{ w. \exists (\overline{y} : \overline{\tau}). (w = v[\overline{y}/\overline{x}]) * P[\overline{y}/\overline{x}] * c \mapsto \text{prot}[\overline{y}/\overline{x}]\} \]

**Session types**

\[ c : \text{chan S} \]

\[ \Gamma \vdash \text{new\_chan}() : \text{chan} S \times \text{chan} \overline{S} \vdash \Gamma \]

\[ \Gamma, x : \text{chan} (!A. S), y : A \vdash \text{send}\ x\ y : \textbf{1} \vdash \]

\[ \Gamma, x : \text{chan} S \]

\[ \Gamma, x : \text{chan} (?A. S) \vdash \text{recv}\ x : A \vdash \]

\[ \Gamma, x : \text{chan} S \]
Example program:

```plaintext
let (c, c’) := new_chan () in
fork {let x := recv c’ in send c’ (x + 2)}; // Service thread
send c 40; recv c // Client thread
```

Dependent separation protocols:

\[ c \leftrightarrow x : \mathbb{Z} \]
\[ c’ \leftrightarrow y : \mathbb{Z} \]

Properties obtained:

\[ □ \text{Program does not crash} \]
\[ □ \text{Program is correct (returns 42)} \]
Example program - via Actris rules

Example program:

```plaintext
let (c, c') := new_chan () in
fork {let x := recv c' in send c' (x + 2)};  // Service thread
send c 40; recv c  // Client thread
```

Dependent separation protocols:

```plaintext
c ↦ ![x: ℤ] ⟨x⟩{True}. ?(y: ℤ) ⟨y⟩{y = (x + 2)}. end and
c' ↦ ?(x: ℤ) ⟨x⟩{True}. ![y: ℤ] ⟨y⟩{y = (x + 2)}. end
```
Example program - via Actris rules

Example program:

```plaintext
let (c, c') := new_chan () in
fork {let x := recv c' in send c' (x + 2)};  // Service thread
send c 40;  recv c  // Client thread
```

Dependent separation protocols:

```plaintext
\n\begin{align*}
& c \rightarrow ! (x: \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ?(y: \mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. \textbf{end} \quad \text{and} \\
& c' \rightarrow ?(x: \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ! (y: \mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. \textbf{end}
\end{align*}
```

Properties obtained:

✓ Program does not crash
Example program - via Actris rules

Example program:

```plaintext
let (c, c') := new-chan () in
fork {let x := recv c' in send c' (x + 2)}; // Service thread
send c 40; recv c // Client thread
```

Dependent separation protocols:

```plaintext
\[ c \rightarrow ! (x: \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ?(y: \mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. \text{end} \]
\[ c' \rightarrow ?(x: \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ! (y: \mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. \text{end} \]
```

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 logical fragments must capture the state of the reliable communication:

- The individual states of the protocols: \( prot_1 \) and \( prot_2 \)
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 $\vec{v}_2$
The Actris Ghost Theory - Logical fragments

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

- The individual states of the protocols: \( \text{prot}_1 \) and \( \text{prot}_2 \)
- 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:**

$$t, u, P, Q ::= \ldots \mid prot\_ctx \chi \vec{v}_1 \vec{v}_2 \mid prot\_own_l \chi prot_1 \mid prot\_own_r \chi prot_2 \mid \ldots$$
The Actris Ghost Theory - Logical fragments

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

- The individual states of the protocols: \( \text{prot}_1 \) and \( \text{prot}_2 \)
- The messages in transit in either direction: \( \vec{v}_1 \) and \( \vec{v}_2 \)

Fragments:

\[
t, u, P, Q ::= \ldots \mid \text{prot}_\text{ctx} \chi \vec{v}_1 \vec{v}_2 \mid \text{prot}_\text{own}_l \chi \text{prot}_1 \mid \text{prot}_\text{own}_r \chi \text{prot}_2 \mid \ldots
\]
The Actris Ghost Theory - Rules

Fragments:

\[ t, u, P, Q ::= \ldots | \text{prot\_ctx} \chi \; \vec{v}_1 \; \vec{v}_2 | \text{prot\_own}_1 \chi \; \text{prot}_1 | \text{prot\_own}_r \chi \; \text{prot}_2 | \ldots \]

NB: only the rules of the left protocol are shown, as the right ones are symmetric
The Actris Ghost Theory - Rules

Fragments:

\[ t, u, P, Q ::= \ldots | \text{prot\_ctx } \chi \ v_1 \ v_2 | \text{prot\_own}_l \ \chi \ \text{prot}_1 | \text{prot\_own}_r \ \chi \ \text{prot}_2 | \ldots \]

Rules:

\[ \text{True} \Rightarrow \exists \chi. \ \text{prot\_ctx } \chi \ \epsilon \ \epsilon^* \ \text{prot\_own}_l \ \chi \ \text{prot}^* \ \text{prot\_own}_r \ \chi \ \overline{\text{prot}} \quad (\text{NEW}) \]

NB: only the rules of the left protocol are shown, as the right ones are symmetric
The Actris Ghost Theory - Rules

Fragments:

\( t, u, P, Q ::= \ldots \mid \text{prot\_ctx} \chi \overrightarrow{v_1} \overrightarrow{v_2} \mid \text{prot\_ownl} \chi \text{\textit{prot}}_1 \mid \text{prot\_ownr} \chi \text{\textit{prot}}_2 \mid \ldots \)

Rules:

\[
\text{True} \Rightarrow \exists \chi. \text{prot\_ctx} \chi \epsilon \epsilon \ast \text{prot\_ownl} \chi \text{\textit{prot}} \ast \text{prot\_ownr} \chi \overline{\text{\textit{prot}}} \quad (\text{NEW})
\]

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

\[
\begin{align*}
\text{HT-vs} \\
\text{\(P \Rightarrow P'\)} & \quad \{P'\} e \{w. Q'\} \\
\overline{\text{\(P\)}} e \{w. Q\}
\end{align*}
\]

NB: only the rules of the left protocol are shown, as the right ones are symmetric
The Actris Ghost Theory - Rules

Fragments:

\[
\begin{align*}
    t, u, P, Q & ::= \ldots \mid \text{prot}_\text{ctx} \chi \overrightarrow{v}_1 \overrightarrow{v}_2 \mid \text{prot}_\text{own}_l \chi \text{prot}_1 \mid \text{prot}_\text{own}_r \chi \text{prot}_2 \mid \ldots
\end{align*}
\]

Rules:

\[
\begin{align*}
    \text{True} & \Rightarrow \exists \chi. \text{prot}_\text{ctx} \chi \epsilon \epsilon * \text{prot}_\text{own}_l \chi \text{prot} * \text{prot}_\text{own}_r \chi \overline{\text{prot}} \quad \text{(NEW)} \\
    \text{prot}_\text{ctx} \chi \overrightarrow{v}_1 \overrightarrow{v}_2 * \text{prot}_\text{own}_l \chi (\mathbf{!} \overrightarrow{x} : \overrightarrow{\tau} \langle \nu \rangle \{ P \}. \text{prot}) * P[\overrightarrow{t}/\overrightarrow{x}] & \Rightarrow \\
    \triangleright |\overrightarrow{v}_2| (\text{prot}_\text{ctx} \chi (\overrightarrow{v}_1 \cdot [\nu[\overrightarrow{t}/\overrightarrow{x}]])) \overrightarrow{v}_2 * \text{prot}_\text{own}_l \chi (\text{prot}[\overrightarrow{t}/\overrightarrow{x}]) \quad \text{(SEND)}
\end{align*}
\]

NB: only the rules of the left protocol are shown, as the right ones are symmetric
The Actris Ghost Theory - Rules

Fragments:

\[ t, u, P, Q ::= \ldots | \text{prot}_\text{ctx} \chi \, \vec{v}_1 \, \vec{v}_2 | \text{prot}_\text{own}_l \chi \, \text{prot}_1 | \text{prot}_\text{own}_r \chi \, \text{prot}_2 | \ldots \]

Rules:

\[
\begin{align*}
\text{True} & \Rightarrow \exists \chi. \, \text{prot}_\text{ctx} \chi \in \epsilon * \text{prot}_\text{own}_l \chi \, \text{prot} * \text{prot}_\text{own}_r \chi \, \overline{\text{prot}} & \quad \text{(NEW)} \\
\text{prot}_\text{ctx} \chi \, \vec{v}_1 \, \vec{v}_2 * \text{prot}_\text{own}_l \chi \left( \Gamma \, x : \bar{\tau} \langle v \rangle \{ P \} \cdot \text{prot} \right) & \Rightarrow \text{P}[\vec{t}/\vec{x}] \\
& \Rightarrow \triangledown_{\vec{v}_2} (\text{prot}_\text{ctx} \chi \left( \vec{v}_1 \cdot [v[t/\vec{x}]] \right) \vec{v}_2) * \text{prot}_\text{own}_l \chi \left( \text{prot}[\vec{t}/\vec{x}] \right) & \quad \text{(SEND)} \\
\text{prot}_\text{ctx} \chi \, \vec{v}_1 \left( \Gamma w \cdot \vec{v}_2 \right) * \text{prot}_\text{own}_l \chi \left( \Delta x : \bar{\tau} \langle v \rangle \{ P \} \cdot \text{prot} \right) & \Rightarrow \triangledown \exists (\vec{y} : \bar{\tau}). \left( w = v[\vec{y}/\vec{x}] \right) * \text{P}[\vec{y}/\vec{x}] \\
& \Rightarrow \text{prot}_\text{ctx} \chi \, \vec{v}_1 \, \vec{v}_2 * \text{prot}_\text{own}_l \chi \left( \text{prot}[\vec{y}/\vec{x}] \right) & \quad \text{(RECV)}
\end{align*}
\]

NB: only the rules of the left protocol are shown, as the right ones are symmetric.
The Actris Ghost Theory - Rules

Fragments:

\[ t, u, P, Q ::= \ldots | \text{prot}_\text{ctx}\, \chi \, \vec{v}_1 \, \vec{v}_2 \mid \text{prot}_\text{own}_l\, \chi \, \text{prot}_1 \mid \text{prot}_\text{own}_r\, \chi \, \text{prot}_2 \mid \ldots \]

Rules:

\[ \text{True} \Rightarrow \exists \chi. \, \text{prot}_\text{ctx}\, \chi \in \epsilon \ast \text{prot}_\text{own}_l\, \chi \, \text{prot} \ast \text{prot}_\text{own}_r\, \chi \, \overline{\text{prot}} \quad \text{(NEW)} \]

\[ \text{prot}_\text{ctx}\, \chi \, \vec{v}_1 \, \vec{v}_2 \ast \text{prot}_\text{own}_l\, \chi \, (\! \vec{x} : \tau \langle \vec{v} \rangle \{ P \}. \text{prot}) \ast P[\vec{t}/\vec{x}] \Rightarrow \]
\[ \triangleright | \vec{v}_2| \left( \text{prot}_\text{ctx}\, \chi \, (\vec{v}_1 \cdot [v[\vec{t}/\vec{x}]])) \, \vec{v}_2) \ast \text{prot}_\text{own}_l\, \chi \, (\text{prot}[\vec{t}/\vec{x}]) \quad \text{(SEND)} \right. \]

\[ \text{prot}_\text{ctx}\, \chi \, \vec{v}_1 \, ([w] \cdot \vec{v}_2) \ast \text{prot}_\text{own}_l\, \chi \, (?\vec{x} : \tau \langle \vec{v} \rangle \{ P \}. \text{prot}) \Rightarrow \]
\[ \triangleright \exists (\vec{y} : \tau). \, (w = v[\vec{y}/\vec{x}]) \ast P[\vec{y}/\vec{x}] \ast \]
\[ \text{prot}_\text{ctx}\, \chi \, \vec{v}_1 \, \vec{v}_2 \ast \text{prot}_\text{own}_l\, \chi \, (\text{prot}[\vec{y}/\vec{x}]) \quad \text{(RCV)} \]

\[ \text{prot}_\text{own}_l\, \chi \, \text{prot} \ast \text{prot} \subseteq \text{prot}' \ast \ast \text{prot}_\text{own}_l\, \chi \, \text{prot}' \quad \text{(SUBLPROTOCOL)} \]

NB: only the rules of the left protocol are shown, as the right ones are symmetric
The Actris Ghost Theory - Rules

Fragments:

\[ t, u, P, Q ::= \ldots | \text{prot}_{\text{ctx}} \chi \vec{v}_1 \vec{v}_2 | \text{prot}_{\text{own}_l} \chi \text{prot}_1 | \text{prot}_{\text{own}_r} \chi \text{prot}_2 | \ldots \]

Rules:

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

\[ \text{True} \sqsubseteq \exists \chi. \text{prot}_{\text{ctx}} \chi \vec{v}_1 \vec{v}_2 \star \text{prot}_{\text{own}_l} \chi \text{prot} \star \text{prot}_{\text{own}_r} \chi \text{prot} \quad (\text{NEW}) \]

\[ \text{prot}_{\text{ctx}} \chi \vec{v}_1 \vec{v}_2 \star \text{prot}_{\text{own}_l} \chi (\! \vec{x} : \vec{\tau} \langle v \rangle \{ P \}. \text{prot}) \star P[\bar{t}/\bar{x}] \Rightarrow \]

\[ \triangleright \vec{v}_2 \: (\text{prot}_{\text{ctx}} \chi (\vec{v}_1 \cdot [v[\bar{t}/\bar{x}]])) \vec{v}_2 \star \text{prot}_{\text{own}_l} \chi (\text{prot}[\bar{t}/\bar{x}]) \quad (\text{SEND}) \]

\[ \text{prot}_{\text{ctx}} \chi \vec{v}_1 ([w] \cdot \vec{v}_2) \star \text{prot}_{\text{own}_l} \chi (?\vec{x} : \vec{\tau} \langle v \rangle \{ P \}. \text{prot}) \Rightarrow \]

\[ \triangleright \exists (\vec{y} : \vec{\tau}). (w = v[\bar{y}/\bar{x}]) \star P[\bar{y}/\bar{x}] \star \]

\[ \text{prot}_{\text{ctx}} \chi \vec{v}_1 \vec{v}_2 \star \text{prot}_{\text{own}_l} \chi (\text{prot}[\bar{y}/\bar{x}]) \quad (\text{RECV}) \]

\[ \text{prot}_{\text{own}_l} \chi \text{prot} \star \text{prot} \sqsubseteq \text{prot}' \star \text{prot}_{\text{own}_l} \chi \text{prot}' \quad (\text{SUBPROTOCOL}) \]

NB: only the rules of the left protocol are shown, as the right ones are symmetric
The Actris Ghost Theory - Rules

Fragments:

\[ t, u, P, Q ::= \ldots \mid \text{prot_ctx } \chi \bar{v}_1 \bar{v}_2 \mid \text{prot_own}_l \chi \text{prot}_1 \mid \text{prot_own}_r \chi \text{prot}_2 \mid \ldots \]

Rules:

\[
\begin{align*}
\text{True} & \Rightarrow \exists \chi. \text{prot_ctx } \chi \in \epsilon * \text{prot_own}_l \chi \text{prot} * \text{prot_own}_r \chi \overline{\text{prot}} && (\text{NEW}) \\
\text{prot_ctx } \chi \bar{v}_1 \bar{v}_2 * \text{prot_own}_l \chi (\forall x : \tau \langle v \rangle \{ P \}. \text{prot}) * P[\bar{t}/\bar{x}] & \Rightarrow \\
& \triangleright | \bar{v}_2| (\text{prot_ctx } \chi (\bar{v}_1 \cdot [v[\bar{t}/\bar{x}]])) \bar{v}_2) * \text{prot_own}_l \chi (\text{prot}[\bar{t}/\bar{x}]) && (\text{SEND}) \\
\end{align*}
\]

\[
\begin{align*}
\text{prot_ctx } \chi \bar{v}_1 ([w] \cdot \bar{v}_2) * \text{prot_own}_l \chi (\forall x : \tau \langle v \rangle \{ P \}. \text{prot}) & \Rightarrow \\
\end{align*}
\]

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

\[
\begin{align*}
\text{HT-step-lb-get} & \quad \text{HT-step-lb-incr} & \quad \text{HT-step-lb-skip} \\
\{P * \times 0\} e \{ w. Q \} & \quad \{P\} e \{ w. Q * \times (n + 1)\} & \quad P_1 \Rightarrow \triangleright^n R \quad \{P_2\} e \{ w. Q * R\} \\
\{P\} e \{ w. Q\} & \quad \{P * \times n\} e \{ w. Q\} & \quad \{P_1 * P_2 * \times n\} e \{ w. Q\} \\
\end{align*}
\]

\[ \text{NB: only the rules of the left protocol are shown, as the right ones are symmetric} \]
The Actris Ghost Theory - Rules

Fragments:

\[ t, u, P, Q ::= \ldots \mid \text{prot}_\text{ctx} \chi \ v_1 \ v_2 \mid \text{prot}_\text{own}_l \ \chi \ \text{prot}_1 \mid \text{prot}_\text{own}_r \ \chi \ \text{prot}_2 \mid \ldots \]

Rules:

\[
\begin{align*}
\text{True} & \Rightarrow \exists \chi. \ \text{prot}_\text{ctx} \ \chi \in \epsilon * \ \text{prot}_\text{own}_l \ \chi \ \text{prot} \ * \ \text{prot}_\text{own}_r \ \chi \ \overline{\text{prot}} \quad \text{(NEW)} \\
\text{prot}_\text{ctx} \ \chi \ \vec{v}_1 \ \vec{v}_2 \ * \ \text{prot}_\text{own}_l \ \chi \ (\exists \vec{x} : \vec{t} \langle \nu \rangle \{P\}. \ \text{prot}) \ * \ P[\vec{t}/\vec{x}] & \Rightarrow \Delta |\vec{v}_2| (\text{prot}_\text{ctx} \ \chi \ (\vec{v}_1 \cdot \nu[\vec{t}/\vec{x}]) \ \vec{v}_2) \ * \ \text{prot}_\text{own}_l \ \chi \ (\text{prot}[\vec{t}/\vec{x}]) \quad \text{(SEND)} \\
\text{prot}_\text{ctx} \ \chi \ \vec{v}_1 \ ([w] \cdot \vec{v}_2) \ * \ \text{prot}_\text{own}_l \ \chi \ (\exists \vec{x} : \vec{t} \langle \nu \rangle \{P\}. \ \text{prot}) & \Rightarrow
\end{align*}
\]

Lower bound of total steps taken (\(\forall n\))

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**
  \[
  \begin{align*}
  \{P \ * \ \vec{x} \ 0\} & \ e \ \{w. \ Q\} \\
  \{P\} & \ e \ \{w. \ Q\}
  \end{align*}
  \]

- **HT-STEP-LB-INCR**
  \[
  \begin{align*}
  \{P\} & \ e \ \{w. \ Q \ * \ \vec{x} \ (n+1)\} \\
  \{P \ * \ \vec{x} \ n\} & \ e \ \{w. \ Q\}
  \end{align*}
  \]

- **HT-STEP-LB-SKIP**
  \[
  \begin{align*}
  P_1 & \Rightarrow \Delta^n R \quad \{P_2\} \ e \ \{w. \ Q \ * \ R\} \\
  \{P_1 \ * \ P_2 \ * \ \vec{x} \ n\} & \ e \ \{w. \ Q\}
  \end{align*}
  \]

NB: Only the rules of the left protocol are shown, as the right ones are symmetric
The Actris Ghost Theory - Rules

Fragments:

\[ t, u, P, Q ::= \ldots | \text{prot\_ctx} \chi v_1 v_2 | \text{prot\_own}_l \chi \text{prot}_1 | \text{prot\_own}_r \chi \text{prot}_2 | \ldots \]

Rules:

\[
\text{True} \Rightarrow \exists \chi. \text{prot\_ctx} \chi \in \epsilon * \text{prot\_own}_l \chi \text{prot} * \text{prot\_own}_r \chi \text{prot} \quad \text{(NEW)}
\]

\[
\text{prot\_ctx} \chi \vec{v}_1 \vec{v}_2 * \text{prot\_own}_l \chi (\vec{x} : \vec{\tau} \langle v \rangle \{ P \}. \text{prot}) * P[t/\vec{x}] \Rightarrow \nabla |\vec{v}_2| (\text{prot\_ctx} \chi (\vec{v}_1 \cdot [v[t/\vec{x}]]) \vec{v}_2) * \text{prot\_own}_l \chi (\text{prot}[t/\vec{x}]) \quad \text{(SEND)}
\]

\[
\text{prot\_ctx} \chi \vec{v}_1 ([w] \cdot \vec{v}_2) * \text{prot\_own}_l \chi (\vec{x} : \vec{\tau} \langle v \rangle \{ P \}. \text{prot}) \Rightarrow
\]

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

\[ \text{HT-step-lb-get} \quad \text{HT-step-lb-incr} \quad \text{HT-step-lb-skip} \]

\[
\{ P * \vec{x} 0 \} e \{ w. Q \} \quad \{ P \} e \{ w. Q * \vec{x} (n+1) \} \quad P_1 \Rightarrow \nabla^n R \quad \{ P_2 \} e \{ w. Q * R \} \quad \{ P_1 * P_2 * \vec{x} n \} e \{ w. Q \}
\]

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

\[
\text{new chan}() := \text{let } (l, r, lk) := (\text{lnil}(), \text{lnil}(), \text{new lock}()) \text{ in } \\
((l, r, lk), (r, l, lk))
\]

\[
\text{send } c \nu
\]

\[
\text{recv } c
\]
Shared memory message passing in HeapLang

We must first provide an implementation of the message passing primitives

\[
\text{new\_chan}() := \text{let} (l, r, lk) := (\text{lnil}(), \text{lnil}(), \text{new\_lock}()) \text{ in }\ \\
((l, r, lk), (r, l, lk))
\]

\[
\text{send} c v := \text{let} (l, r, lk) := c \text{ in }\ \\
\text{acquire} lk; \quad \text{lsnoc} l v; \quad \text{release} lk
\]

\[
\text{recv} c
\]
Shared memory message passing in HeapLang

We must first provide an implementation of the message passing primitives

```plaintext
new_chan () := let (l, r, lk) := (lnil (), lnil (), new_lock ()) in
((l, r, lk), (r, l, lk))

send c v := let (l, r, lk) := c in
acquire lk;
  lsnoc l v;
release lk

recv c := match (try_recv c) with
  inj1 () ⇒ recv c
  | inj2 v ⇒ v
end

try_recv c := let (l, r, lk) := c in
acquire lk;
  let ret := (if (lisnil r) then (inj1 ()) else (inj2 (lpop r))) in
release lk; ret
```
Defining the channel endpoint ownership

Defining the channel endpoint ownership \( c \mapsto prot \)
Defining the channel endpoint ownership

Defining the channel endpoint ownership $c \mapsto prot$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state.
Defining the channel endpoint ownership

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

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\text{prot}\_\text{own}_l \chi \text{prot} / \text{prot}\_\text{own}_r \chi \text{prot}$
Defining the channel endpoint ownership

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

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\text{prot}_\text{own}_l \chi \text{prot} / \text{prot}_\text{own}_r \chi \text{prot}$
  - Include the shared protocol context: $\text{prot}_{\text{ctx}} \chi \vec{v}_1 \vec{v}_2$
Defining the channel endpoint ownership

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

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\text{prot\_own}_l \chi prot / \text{prot\_own}_r \chi prot$
  - Include the shared protocol context: $\text{prot\_ctx} \chi \vec{v}_1 \vec{v}_2$
  - 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 $c \mapsto \textit{prot}$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\text{prot\_own}_l \chi \textit{prot} / \text{prot\_own}_r \chi \textit{prot}$
  - Include the shared protocol context: $\text{prot\_ctx} \chi \vec{v}_1 \vec{v}_2$
  - Include the step lower bound for each logical buffer: $\bar{x} |\vec{v}_1|$ and $\bar{x} |\vec{v}_2|$

- **Implementation-specific physical state**
  - Capture the structure of the channel abstraction $c$
Defining the channel endpoint ownership

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

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\text{prot}_\text{own}_l \chi prot / \text{prot}_\text{own}_r \chi prot$
  - Include the shared protocol context: $\text{prot}_\text{ctx} \chi \vec{v}_1 \vec{v}_2$
  - Include the step lower bound for each logical buffer: $\overline{x} |\vec{v}_1|$ and $\overline{x} |\vec{v}_2|$

- **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 $c \mapsto prot$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

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

- **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 $c \mapsto prot$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\text{prot}_\text{own}_l \chi prot / \text{prot}_\text{own}_r \chi prot$
  - Include the shared protocol context: $\text{prot}_\text{ctx} \chi \vec{v}_1 \vec{v}_2$
  - Include the step lower bound for each logical buffer: $\chi |\vec{v}_1| \text{ and } \chi |\vec{v}_2|$

- **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 $c \mapsto prot$ requires connecting the implementation-agnostic logical state with the implementation-specific physical state:

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\text{prot}_\text{own}_l \chi prot / \text{prot}_\text{own}_r \chi prot$
  - Include the shared protocol context: $\text{prot}_\text{ctx} \chi \vec{v}_1 \vec{v}_2$
  - Include the step lower bound for each logical buffer: $\exists |\vec{v}_1|$ and $\exists |\vec{v}_2|$

- **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: $\text{isList} l \vec{v}_1 / \text{isList} r \vec{v}_2$
  - Include a means of synchronisation between the two endpoints
Defining the channel endpoint ownership

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

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\text{prot\_own}_l \chi \text{prot} / \text{prot\_own}_r \chi \text{prot}$
  - Include the shared protocol context: $\text{prot\_ctx}_\chi \vec{v}_1 \vec{v}_2$
  - Include the step lower bound for each logical buffer: $\chi |\vec{v}_1|$ and $\chi |\vec{v}_2|$

- **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: $\text{isList} l \vec{v}_1 / \text{isList} r \vec{v}_2$
  - Include a means of synchronisation between the two endpoints

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

$$\text{HT-LNIL}$$

{\{\text{True}\} \text{lnil} \{l.\text{isList} l [\quad]\}}

$$\text{HT-LSNOC}$$

{\{\text{isList} l \vec{x} \ast l \times v\} \text{lsnoc} l v \{\text{isList} l (\vec{x} \cdot [x])\}}
Defining the channel endpoint ownership

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

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $prot_{own_l} \chi prot / prot_{own_r} \chi prot$
  - Include the shared protocol context: $prot_{ctx} \chi \vec{v}_1 \vec{v}_2$
  - Include the step lower bound for each logical buffer: $\chi |\vec{v}_1|$ and $\chi |\vec{v}_2|$

- **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: $isList l \vec{v}_1 / isList r \vec{v}_2$
  - Include a means of synchronisation between the two endpoints: $is\_lock lk R$
Defining the channel endpoint ownership

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

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\text{prot}_\text{own}_l \chi \text{prot} / \text{prot}_\text{own}_r \chi \text{prot}$
  - Include the shared protocol context: $\text{prot}_\text{ctx} \chi \vec{v}_1 \vec{v}_2$
  - Include the step lower bound for each logical buffer: $\exists |\vec{v}_1|$ and $\exists |\vec{v}_2|$

- **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: $\text{isList} l \vec{v}_1 / \text{isList} r \vec{v}_2$
  - Include a means of synchronisation between the two endpoints: $\text{is_lock} \ l k \ R$

---

Lock ownership ($\text{is_lock} \ l k \ R$) asserts that the lock $lk$ governs the proposition $R$

\[ \text{HT-acquire} \]
\[
\{ \text{is_lock} \ l k \ R \} \text{ acquire } l k \ { \{ R \} } \]

\[ \text{HT-release} \]
\[
\{ \text{is_lock} \ l k \ R \ast \ R \} \text{ release } l k \ { \{ \text{True} \} } \]
Defining the channel endpoint ownership

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

- **Implementation-agnostic logical state**
  - Assert ownership of the respective protocol: $\text{prot}_\text{own}_l \, \chi \, \text{prot} / \, \text{prot}_\text{own}_r \, \chi \, \text{prot}$
  - Include the shared protocol context: $\text{prot}_\text{ctx} \, \chi \, \vec{v}_1 \, \vec{v}_2$
  - Include the step lower bound for each logical buffer: $\chi \, |\vec{v}_1|$ and $\chi \, |\vec{v}_2|$

- **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: $\text{isList} \, l \, \vec{v}_1 / \text{isList} \, r \, \vec{v}_2$
  - Include a means of synchronisation between the two endpoints: $\text{is\_lock} \, lk \, R$

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

$$
\begin{align*}
  c \mapsto prot & \triangleq \exists \chi, l, r, lk. \left( (c = (l, r, lk) \, * \, \text{prot}_\text{own}_l \, \chi \, \text{prot}) \, \lor \right. \\
  & \left. (c = (r, l, lk) \, * \, \text{prot}_\text{own}_r \, \chi \, \text{prot}) \right) \, * \\
  & \text{is\_lock} \, lk \, (\exists \vec{v}_1 \, \vec{v}_2. \, \text{isList} \, l \, \vec{v}_1 \, / \, \text{isList} \, r \, \vec{v}_2 \, * \\
  & \quad \text{prot}_\text{ctx} \, \chi \, \vec{v}_1 \, \vec{v}_2 \, * \, \chi \, |\vec{v}_1| \, * \, \chi \, |\vec{v}_2|)
\end{align*}
$$
We wish to prove:

\[ \{\text{True}\} \text{new\_chan} () \{w. \exists c_1, c_2. w = (c_1, c_2) \ast c_1 \rightarrow \text{prot} \ast c_2 \rightarrow \overline{\text{prot}}\} \]

It follows almost directly from the rule:

\[ \text{True} \Rightarrow \exists \chi. \text{prot\_ctx} \chi \in \epsilon \ast \text{prot\_own}_l \chi \text{ prot} \ast \text{prot\_own}_r \chi \overline{\text{prot}} \]

And the definition of the channel endpoint ownership:

\[ c \rightarrow \text{prot} \triangleq \exists \chi, l, r, lk. \left( (c = (l, r, lk) \ast \text{prot\_own}_l \chi \text{ prot} ) \lor (c = (r, l, lk) \ast \text{prot\_own}_r \chi \text{ prot} ) \right) \ast \text{is\_lock} \ lk \ (\exists \vec{v}_1 \vec{v}_2. \text{isList} \ l \ \vec{v}_1 \ast \text{isList} \ r \ \vec{v}_2 \ast \text{prot\_ctx} \chi \ \vec{v}_1 \vec{v}_2 \ast \vec{v}_1 \ast \vec{v}_2 \ast |\vec{v}_1| \ast |\vec{v}_2|) \]
We wish to prove:

\[
\{ c \rightarrow ! \vec{x} : \vec{\tau} \langle v \rangle \{ P \}. \text{prot} * P[\vec{t}/\vec{x}] \} \text{send } c (v[\vec{t}/\vec{x}]) \{ c \rightarrow \text{prot}[\vec{t}/\vec{x}] \}
\]

It follows almost directly from the rule:

\[
\text{prot}_\text{ctx} \ \chi \ \vec{v}_1 \ \vec{v}_2 * \text{prot}_\text{own}_l \ \chi \ ( ! \vec{x} : \vec{\tau} \langle v \rangle \{ P \}. \text{prot} ) * P[\vec{t}/\vec{x}] \Rightarrow
\]
\[
\triangleright | \vec{v}_2 | ( \text{prot}_\text{ctx} \ \chi ( \vec{v}_1 \cdot [v[\vec{t}/\vec{x}]] ) \ \vec{v}_2 ) * \text{prot}_\text{own}_l \ \chi ( \text{prot}[\vec{t}/\vec{x}] )
\]

And the definition of the channel endpoint ownership:

\[
c \rightarrow \text{prot} \triangleq \exists \chi, l, r, \text{lk}. \ ((c = (l, r, \text{lk}) * \text{prot}_\text{own}_l \ \chi \ \text{prot}) \lor
\]
\[
((c = (r, l, \text{lk}) * \text{prot}_\text{own}_r \ \chi \ \text{prot}) * \text{is_lock} \ \text{lk} ( \exists \vec{v}_1 \ \vec{v}_2 . \text{isList} \ l \ \vec{v}_1 \ * \text{isList} \ r \ \vec{v}_2 * \text{prot}_\text{ctx} \ \chi \ \vec{v}_1 \ \vec{v}_2 * \vec{v}_1 | * \vec{v}_2 | )
\]
We wish to prove:

\[
\{ c \rightarrow ?x:\tau\langle v\rangle\{ P \}. prot \} \xrightarrow{recv} \{ w. \exists y. w = v[y/x] * c \rightarrow \text{prot}[y/x] * P[y/x] \}
\]

It follows almost directly from the rule:

\[
\text{prot\_ctx}\chi\vec{v}_1\left(\left([w] \cdot \vec{v}_2\right) * \text{prot\_own}_l\chi\left(?x:\tau\langle v\rangle\{ P \}. prot\right)\right) \Rightarrow \\
\triangleright \exists (\vec{y}:\vec{\tau}). \left( w = v[y/x] \right) * P[y/x] * \\
\text{prot\_ctx}\chi\vec{v}_1\vec{v}_2 * \text{prot\_own}_l\chi\left(\text{prot}[y/x]\right)
\]

And the definition of the channel endpoint ownership:

\[
c \rightarrow \text{prot} \triangleq \exists \chi, l, r, lk. \left( \left( c = (l, r, lk) * \text{prot\_own}_l\chi\text{prot} \right) \lor \left( c = (r, l, lk) * \text{prot\_own}_r\chi\text{prot} \right) \right) * \\
\text{is\_lock}\ lk\left( \exists \vec{v}_1\vec{v}_2.\text{isList}\ l\ \vec{v}_1 * \text{isList}\ r\ \vec{v}_2 * \\
\text{prot\_ctx}\chi\vec{v}_1\vec{v}_2 * \not Jal\vec{v}_1 | \not Jal\vec{v}_2 \right)
\]
1. Dependent separation protocols
2. Actris Rules
3. Actris Ghost Theory
4. Mechanisation of Actris
Mechanisation of Actris

**Dependent separation protocols:**

- Define the type of $prot$ using Iris’s recursive domain equation solver

**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 $c \mapsto prot$ using the Actris ghost theory
- Prove the Actris rules as lemmas in Iris, using the ghost theory rules
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
  - !\bar{x}:(\bar{\tau}\langle v \rangle\{P\}, prot, \overline{prot}, and prot_1 \sqsubseteq 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 $prot$
  - $!\vec{x} : \tau \langle v \rangle \{P\}. \overline{prot}, \overline{prot}$, and $prot_1 \sqsubseteq prot_2$

**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 $prot$
  - $! \vec{x}: \vec{\tau}(\langle v \rangle \{P\}. prot, \overline{prot},$ 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
Mechanisation of Actris

Dependent separation protocols:
- Define the type of \( prot \) using Iris’s recursive domain equation solver
- Define constructors, operations, and relations on \( prot \)
  - ! \( x : \tau \langle v \rangle \{ P \} \). \( prot, \overline{prot} \), 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
- 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 \(prot\)
  - !\(\vec{x}:\vec{\tau}\langle v \rangle \{ P \}\). \(prot\), \(\overline{prot}\), 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
- 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 $prot$
  - $\forall \vec{x} : \vec{\tau} \langle v \rangle \{ P \}. \overline{prot}, \overline{prot}$, 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
- 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 $c \mapsto prot$ 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 $prot$
  - $!\vec{x}:\tau \langle v \rangle \{ P \}. prot, \overline{prot}$, 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
- 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 $c \mapsto prot$ using the Actris ghost theory
- Prove the Actris rules as lemmas in Iris, using the ghost theory rules
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)

Publications

Actris: Session-Type Based Reasoning in Separation Logic

JONAS KASTBERG HINRICHSEN, IT University of Copenhagen, Denmark
JEPPE BENGTSON, IT University of Copenhagen, Denmark
ROBERT KREIBEERS, Chalmers 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 useful level of abstraction by proving functional correctness of a variety of examples, including a distributed range test, a distributed load-balancing mapper, and a version 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 mechanized the theory of Actris, together with tactics for symbolic execution of programs, as well as 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:

1 INTRODUCTION

Message-passing programs are ubiquitous in modern computer systems, emphasizing the importance of these functional connectors. Preprocessing languages like Ptolemy, Pretzel, and Co have been shown to be useful abstractions for reasoning about message-passing programs. In this paper, we present Actris, a protocol mechanism for reasoning about message passing in the presence of other concurrency paradigms.

Machine-Checked Semantic Session Typing

Jonas Kastberg Hinrichsen, IT University of Copenhagen, Denmark
Bjarne Sandemose, IT University of Copenhagen, Denmark

Certified Programs and Proofs Conference 2021 [CPP’21]

Abstract

Machine-checked proofs of semantic correctness for message-passing programs have been shown to be a useful abstraction for reasoning about these programs. The main challenge in such proofs is to connect the machine-checked proof with a high-level semantics of the program. In this paper, we present a machine-checked proof of semantic correctness for a program that uses message passing and session typing. We prove the correctness of a program that sends messages between processes using a session type. We also prove the correctness of a program that receives messages from processes using a session type.

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

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

ACM Reference Format:

2. Session types are used to enforce a strict structure of channel connectivity. While conventional session type systems on the type level enforce this structure, they sometimes require some flexibility that can only be achieved by using a variety of mechanisms such as session typing on the type level. In this paper, we consider the following challenges.

1. There are many extensions of session types with e.g., polymorphism [2], subtyping [3], and subtyping [4].

2. Type safety has been proven for many extensions of session types, existing proofs cannot be readily adapted to new extensions.

3. We believe that this guideline is essential for the success of the Iris framework. Without an additional level of abstraction, Iris cannot be used to reason about programs that are written in the Iris framework.

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

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

ACM Reference Format:
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
Thank you. 

µ rec. ?(q : Question) ⟨q⟩ {AboutActris q}.

!(a : Answer) ⟨a⟩ {Insightful q a}. rec