

# Actris: Session-Type Based Reasoning in Separation Logic

**Jonas Kastberg Hinrichsen**  
**IT University of Copenhagen**

30. November 2020  
Aarhus University

# The actor model and message passing

**Principled way of writing concurrent programs**

# The actor model and message passing

## **Principled way of writing concurrent programs**

- ▶ Isolation of concurrent behaviour

# The actor model and message passing

## **Principled way of writing concurrent programs**

- ▶ Isolation of concurrent behaviour
- ▶ Threads as services and clients

# The actor model and message passing

## **Principled way of writing concurrent programs**

- ▶ Isolation of concurrent behaviour
- ▶ Threads as services and clients
- ▶ Used in Erlang, Elixir, Go, Java, Scala, F# and C#

# The actor model and message passing

## Principled way of writing concurrent programs

- ▶ Isolation of concurrent behaviour
- ▶ Threads as services and clients
- ▶ Used in Erlang, Elixir, Go, Java, Scala, F# and C#

## Message passing primitives

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

# The actor model and message passing

## Principled way of writing concurrent programs

- ▶ Isolation of concurrent behaviour
- ▶ Threads as services and clients
- ▶ Used in Erlang, Elixir, Go, Java, Scala, F# and C#

## Message passing primitives

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

**Example:** `let (c, c') = new_chan () in  
fork {let x = recv c' in send c' (x + 2)};  
send c 40; recv c`

# The actor model and message passing

## Principled way of writing concurrent programs

- ▶ Isolation of concurrent behaviour
- ▶ Threads as services and clients
- ▶ Used in Erlang, Elixir, Go, Java, Scala, F# and C#

## Message passing primitives

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

**Example:** `let (c, c') = new_chan () in  
fork {let x = recv c' in send c' (x + 2)};  
send c 40; recv c`

## Many variants of message passing exist

We consider: asynchronous, order-preserving and reliable



## **Message passing is not a silver bullet for concurrency**

“We studied 15 large, mature, and actively maintained actor programs written in Scala and found that 80% of them mix the actor model with another concurrency model.” [ [Tasharofi et al., ECOOP'13](#) ]

## **Message passing is not a silver bullet for concurrency**

“We studied 15 large, mature, and actively maintained actor programs written in Scala and found that 80% of them mix the actor model with another concurrency model.” [ [Tasharofi et al., ECOOP'13](#) ]

**Problem:** No existing solution for dependent high-level actor-based reasoning in combination with existing concurrency models for functional correctness

## Message passing is not a silver bullet for concurrency

“We studied 15 large, mature, and actively maintained actor programs written in Scala and found that 80% of them mix the actor model with another concurrency model.” [ [Tasharofi et al., ECOOP'13](#) ]

**Problem:** No existing solution for dependent high-level actor-based reasoning in combination with existing concurrency models for functional correctness

- ▶ **Dependent:** dependency on previously communicated messages

## Message passing is not a silver bullet for concurrency

“We studied 15 large, mature, and actively maintained actor programs written in Scala and found that 80% of them mix the actor model with another concurrency model.” [ [Tasharofi et al., ECOOP'13](#) ]

**Problem:** No existing solution for dependent high-level actor-based reasoning in combination with existing concurrency models for functional correctness

- ▶ **Dependent:** dependency on previously communicated messages
- ▶ **High-level:** communication of references, channels and higher-order functions

## Key Idea

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

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

**Session types** [ [Honda et al., ESOP'98](#) ]

- ▶ Type system for channel endpoints
- ▶ Example: `!Z. ?Z. end`
- ▶ Ensures safety and session fidelity

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

**Session types** [ [Honda et al., ESOP'98](#) ]

- ▶ Type system for channel endpoints
- ▶ Example:  $!Z. ?Z. \text{end}$
- ▶ Ensures safety and session fidelity

**Iris's concurrent separation logic** [ [Jung et al., POPL'15](#) ]

- ▶ Logic for reasoning about concurrent programs with mutable state
- ▶ Example:  $\{\ell \mapsto v\} \ell \leftarrow w \{\ell \mapsto w\}$
- ▶ Supports high-level concurrency reasoning mechanisms
- ▶ Ensures functional correctness

**Actris:** A concurrent separation logic for proving *functional correctness* of programs that combine *message passing* with other programming and concurrency paradigms

- ▶ Introducing *dependent separation protocols*
- ▶ Integration with Iris and its existing concurrency mechanisms
- ▶ Verification of feature-heavy programs including a variant of map-reduce
- ▶ Full mechanization in Coq (<https://gitlab.mpi-sws.org/iris/actris/>)



# Features of dependent separation protocols

Specification and proof system for message passing that allows

- ▶ **Resources:** sending references
- ▶ **Higher-order:** sending function closures
- ▶ **Delegation:** sending channels over channels
- ▶ **Dependent:** dependency on previous messages
- ▶ **Recursion:** looping protocols
- ▶ **Choice:** diverging protocols
- ▶ **Manifest sharing:** concurrent sharing of channel endpoints
- ▶ **Subprotocols:** weakening mechanism for added flexibility

# Features of dependent separation protocols

Specification and proof system for message passing that allows

- ▶ **Resources**: sending references
- ▶ **Higher-order**: sending function closures
- ▶ **Delegation**: sending channels over channels
- ▶ **Dependent**: dependency on previous messages
- ▶ **Recursion**: looping protocols
- ▶ **Choice**: diverging protocols
- ▶ **Manifest sharing**: concurrent sharing of channel endpoints
- ▶ **Subprotocols**: weakening mechanism for added flexibility

# Actris

joint work with

Jesper Bengtson, IT University of Copenhagen

Robbert Krebbers, Radboud University

# Tour of Actris - Goal

**Language:** ML-like language extended with concurrency, state and message passing

$$e \in \text{Expr} ::= v \mid x \mid \text{rec } f \ x = e \mid e_1(e_2) \mid \text{fork } \{e\} \mid \text{ref } (e) \mid !e \mid e_1 \leftarrow e_2 \mid \\ \text{new\_chan } () \mid \text{send } e_1 \ e_2 \mid \text{recv } e \mid \dots$$

# Tour of Actris - Goal

**Language:** ML-like language extended with concurrency, state and message passing

$$e \in \text{Expr} ::= v \mid x \mid \text{rec } f \ x = e \mid e_1(e_2) \mid \text{fork } \{e\} \mid \text{ref } (e) \mid !e \mid e_1 \leftarrow e_2 \mid \\ \text{new\_chan } () \mid \text{send } e_1 \ e_2 \mid \text{recv } e \mid \dots$$

**Example program:**

```
let (c, c') = new_chan () in
fork {let x = recv c' in send c' (x + 2)};
send c 40; recv c
```

# Tour of Actris - Goal

**Language:** ML-like language extended with concurrency, state and message passing

$$e \in \text{Expr} ::= v \mid x \mid \text{rec } f \ x = e \mid e_1(e_2) \mid \text{fork } \{e\} \mid \text{ref } (e) \mid !e \mid e_1 \leftarrow e_2 \mid \\ \text{new\_chan } () \mid \text{send } e_1 \ e_2 \mid \text{recv } e \mid \dots$$

**Example program:**

```
let (c, c') = new_chan () in
fork {let x = recv c' in send c' (x + 2)};
send c 40; recv c
```

**Goal:** prove that returned value is 42

# Session types

## Symbols

$$\begin{array}{l|l} S ::= !A. S & \\ \quad ?A. S & \\ \quad \text{end} & \dots \end{array}$$

## Example

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

## Duality

$$\begin{array}{l} \overline{!A. S} = ?A. \overline{S} \\ \overline{?A. S} = !A. \overline{S} \\ \overline{\text{end}} = \text{end} \end{array}$$

## Usage

$c : \text{chan } S$

## Rules

$\text{new\_chan } () : \text{chan } S \times \text{chan } \overline{S}$

$\text{send} : (\text{chan } (!A. S) \times A) \multimap \text{chan } S$

$\text{recv} : \text{chan } (?A. S) \multimap (A \times \text{chan } S)$

### Example program:

```
let (c, c') = new_chan () in  
fork {let x = recv c' in send c' (x + 2)};  
send c 40; recv c
```



## Tour of Actris - Type checked

### Example program:

```
let (c, c') = new_chan () in  
fork {let x = recv c' in send c' (x + 2)};  
send c 40; recv c
```

### Session types:

$c$  : chan (!Z. ?Z. end)      and  
 $c'$  : chan (?Z. !Z. end)

# Tour of Actris - Type checked

## Example program:

```
let (c, c') = new_chan () in
fork {let x = recv c' in send c' (x + 2)};
send c 40; recv c
```

## Session types:

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

## Properties obtained:

- ✓ Safety / session fidelity
- ✗ Functional correctness

# Dependent separation protocols - Definitions

	<u>Dependent separation protocols</u>	<u>Session types</u>
<b>Symbols</b>	$  \begin{array}{l}  prot ::= !\vec{x}:\vec{\tau}\langle v\rangle\{P\}. prot \quad   \\  \qquad \qquad ?\vec{x}:\vec{\tau}\langle v\rangle\{P\}. prot \quad   \\  \qquad \qquad \text{end}  \end{array}  $	$  \begin{array}{l}  S ::= !A. S \quad   \\  \qquad \qquad ?A. S \quad   \\  \qquad \qquad \text{end} \quad   \dots  \end{array}  $
<b>Example</b>	$!(x:\mathbb{Z})\langle x\rangle\{\text{True}\}.?(y:\mathbb{Z})\langle y\rangle\{y = (x + 2)\}.\text{end}$	$!Z. ?Z. \text{end}$
<b>Duality</b>	$  \begin{array}{l}  \overline{!\vec{x}:\vec{\tau}\langle v\rangle\{P\}. prot} = ?\vec{x}:\vec{\tau}\langle v\rangle\{P\}.\overline{prot} \\  \overline{?\vec{x}:\vec{\tau}\langle v\rangle\{P\}. prot} = !\vec{x}:\vec{\tau}\langle v\rangle\{P\}.\overline{prot} \\  \overline{\text{end}} = \text{end}  \end{array}  $	$  \begin{array}{l}  \overline{!A. S} = ?A.\overline{S} \\  \overline{?A. S} = !A.\overline{S} \\  \overline{\text{end}} = \text{end}  \end{array}  $
<b>Usage</b>	$c \multimap prot$	$c : \text{chan } S$

# Dependent separation protocols - Rules

	<u>Dependent separation protocols</u>	<u>Session types</u>
<b>New</b>	$\{\text{True}\}$ $\text{new\_chan } ()$ $\{(c, c'). c \multimap \text{prot} * c' \multimap \overline{\text{prot}}\}$	$\text{new\_chan } () : \text{chan } S \times \text{chan } \overline{S}$
<b>Send</b>	$\{c \multimap !\vec{x} : \vec{\tau} \langle v \rangle \{P\}. \text{prot} * P[\vec{t}/\vec{x}]\}$ $\text{send } c (v[\vec{t}/\vec{x}])$ $\{c \multimap \text{prot}[\vec{t}/\vec{x}]\}$	$\text{send} : (\text{chan } (!A. S) \times A) \multimap \text{chan } S$
<b>Recv</b>	$\{c \multimap ?\vec{x} : \vec{\tau} \langle v \rangle \{P\}. \text{prot}\}$ $\text{recv } c$ $\{w. \exists(\vec{y} : \vec{\tau}). (w = v[\vec{y}/\vec{x}]) * \\ c \multimap \text{prot}[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\}$	$\text{recv} : \text{chan } (?A. S) \multimap (A \times \text{chan } S)$

## Example program:

```
let (c, c') = new_chan () in  
fork {let x = recv c' in send c' (x + 2)};  
send c 40; recv c
```

## Example program:

```
let (c, c') = new_chan () in
fork {let x = recv c' in send c' (x + 2)};
send c 40; recv c
```

## Dependent separation protocols:

$$\begin{aligned} c &\mapsto ! (x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ? (y : \mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. \text{end} && \text{and} \\ c' &\mapsto ? (x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ! (y : \mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. \text{end} \end{aligned}$$

## Example program:

```
let (c, c') = new_chan () in
fork {let x = recv c' in send c' (x + 2)};
send c 40; recv c
```

## Dependent separation protocols:

$$\begin{aligned} c &\mapsto !(x:\mathbb{Z}) \langle x \rangle \{ \text{True} \}. ?(y:\mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. \text{end} && \text{and} \\ c' &\mapsto ?(x:\mathbb{Z}) \langle x \rangle \{ \text{True} \}. !(y:\mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. \text{end} \end{aligned}$$

## Properties obtained:

- ✓ Safety / session fidelity
- ✓ Functional correctness

# Tour of Actris - References

## Example program:

```
let (c, c') = new_chan () in  
fork {let  $\ell$  = recv c' in  $\ell \leftarrow (!\ell + 2)$ ; send c' ()};  
let  $\ell$  = ref 40 in send c  $\ell$ ; recv c; ! $\ell$ 
```



# Tour of Actris - References

## Example program:

```
let (c, c') = new_chan () in
fork {let  $\ell$  = recv c' in  $\ell \leftarrow (!\ell + 2)$ ; send c' ()};
let  $\ell$  = ref 40 in send c  $\ell$ ; recv c; ! $\ell$ 
```


## Dependent separation protocols:

$$c \mapsto !(l:\text{Loc}) (x:\mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ? \langle () \rangle \{ \ell \mapsto (x + 2) \}. \text{end} \quad \text{and}$$
$$c' \mapsto ?(l:\text{Loc}) (x:\mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ! \langle () \rangle \{ \ell \mapsto (x + 2) \}. \text{end}$$

# Tour of Actris - References

## Example program:

```
let (c, c') = new_chan () in
fork {let l = recv c' in l ← (!l + 2); send c' ()};
let l = ref 40 in send c l; recv c; !l
```



$\{\text{True}\} \text{ref } v \{l. l \mapsto v\}$

## Dependent separation protocols:

$c \mapsto !(l:\text{Loc}) (x:\mathbb{Z}) \langle l \rangle \{l \mapsto x\}. ?\langle () \rangle \{l \mapsto (x + 2)\}. \text{end}$       and

$c' \mapsto ?(l:\text{Loc}) (x:\mathbb{Z}) \langle l \rangle \{l \mapsto x\}. !\langle () \rangle \{l \mapsto (x + 2)\}. \text{end}$

# Tour of Actris - References

## Example program:

```
let (c, c') = new_chan () in
fork {let l = recv c' in l ← (!l + 2); send c' ()};
let l = ref 40 in send c l; recv c; !l
```

$\{\text{True}\} \text{ref } v \{l. l \mapsto v\}$

$\{l \mapsto v\} !l \{w. w = v \wedge l \mapsto v\}$

## Dependent separation protocols:

$c \mapsto !(l:\text{Loc}) (x:\mathbb{Z}) \langle l \rangle \{l \mapsto x\}. ?\langle () \rangle \{l \mapsto (x + 2)\}. \text{end}$       and  
 $c' \mapsto ?(l:\text{Loc}) (x:\mathbb{Z}) \langle l \rangle \{l \mapsto x\}. !\langle () \rangle \{l \mapsto (x + 2)\}. \text{end}$

## Example - Recursion

### Example program:

```
let (c, c') = new_chan () in
fork {let go () = (let x = recv c' in send c' (x + 2); go ()) in go ()};
send c 18; let x = recv c in
send c 20; let y = recv c in x + y
```

## Example - Recursion

### Example program:

```
let (c, c') = new_chan () in
fork {let go () = (let x = recv c' in send c' (x + 2); go ()) in go ()};
send c 18; let x = recv c in
send c 20; let y = recv c in x + y
```

### Dependent separation protocols:

$$\begin{aligned} c &\rightsquigarrow \mu rec. ! (x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ? (y : \mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. rec & \text{and} \\ c' &\rightsquigarrow \mu rec. ? (x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ! (y : \mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. rec \end{aligned}$$

## Example - Recursion

### Example program:

```
let (c, c') = new_chan () in
fork {let go () = (let x = recv c' in send c' (x + 2); go ()) in go ()};
send c 18; let x = recv c in
send c 20; let y = recv c in x + y
```

### Dependent separation protocols:

$$\begin{aligned} c &\mapsto \mu rec. ! (x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ? (y : \mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. rec && \text{and} \\ c' &\mapsto \mu rec. ? (x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ! (y : \mathbb{Z}) \langle y \rangle \{ y = (x + 2) \}. rec \end{aligned}$$

### Proof:

- ▶ Client thread: follows immediately from Actris's rules
- ▶ Service thread: follows immediately using Löb induction

## Example - Locks

**Example program:**

```
let (c, c') = new_chan () in
fork {
  let lk = new_lock () in
  fork { acquire lk; send c' 21; release lk };
  acquire lk; send c' 21; release lk
};
recv c + recv c
```

## Example - Locks

### Example program:

```
let (c, c') = new_chan () in
fork {
  let lk = new_lock () in
  fork { acquire lk; send c' 21; release lk };
  acquire lk; send c' 21; release lk
};
recv c + recv c
```

### Dependent separation protocols:

```
lock_prot (n :  $\mathbb{N}$ )  $\triangleq$ 
  if n = 0 then end
  else ?⟨21⟩{True}.lock_prot (n - 1)
```



## Example - Locks

**Example program:**

```
let (c, c') = new_chan () in
fork {
  let lk = new_lock () in
  fork {acquire lk; send c' 21; release lk};
  acquire lk; send c' 21; release lk
};
recv c + recv c
```

**Dependent separation protocols:**

```
lock_prot (n : ℕ)  $\triangleq$ 
  if n = 0 then end
  else ?⟨21⟩{True}.lock_prot (n - 1)
```

$$\begin{array}{l} c \multimap \text{lock\_prot } 2 \\ c' \multimap \overline{\text{lock\_prot } 2} \end{array} \quad \text{and}$$

## Example - Locks

Example program:

```
let (c, c') = new_chan () in
fork {
  let lk = new_lock () in
  fork {acquire lk; send c' 21; release lk};
  acquire lk; send c' 21; release lk
};
recv c + recv c
```

Dependent separation protocols:

$\text{lock\_prot } (n : \mathbb{N}) \triangleq$   
if  $n = 0$  then end  
else  $? \langle 21 \rangle \{ \text{True} \}. \text{lock\_prot } (n - 1)$

$c \mapsto \text{lock\_prot } 2$       and  
 $c' \mapsto \overline{\text{lock\_prot } 2}$

Hoare triple for critical section:

$\left\{ \begin{array}{l} \text{is\_lock } lk \ (\exists n. c' \mapsto \overline{\text{lock\_prot } n} *) \\ \left[ \bullet n : \text{Auth}(\mathbb{N}) \right]^\gamma * \left[ \circ 1 : \text{Auth}(\mathbb{N}) \right]^\gamma \end{array} \right\}$   
acquire  $lk$ ; send  $c' 21$ ; release  $lk$   
 $\{ \text{True} \}$

Paper [POPL'20]: [https://itu.dk/people/jkas/papers/actris\\_popl.pdf](https://itu.dk/people/jkas/papers/actris_popl.pdf)

Mechanisation: <https://gitlab.mpi-sws.org/iris/actris/-/tree/popl20>

## Actris: Session-Type Based Reasoning in Separation Logic

JONAS KASTBERG HINRICHSSEN, IT University of Copenhagen, Denmark

JESPER BENGTON, 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>

# Actris 2.0

joint work with

Jesper Bengtson, IT University of Copenhagen

Robbert Krebbers, Radboud University

## Problem 1 - Lack of expressivity

**Actris 1.0 does not take advantage of the asynchronous semantics**

## Problem 1 - Lack of expressivity

**Actris 1.0 does not take advantage of the asynchronous semantics**

- ▶ Bi-directional buffers allows messages in transit in both directions

## Problem 1 - Lack of expressivity

### Actris 1.0 does not take advantage of the asynchronous semantics

- Bi-directional buffers allows messages in transit in both directions

### Example Program

```
let (c, c') = new_chan () in
fork {send c' 20; let x = recv c' in send c' (x + 2)};
send c 20;
let x = recv c in
let y = recv c in x + y
```

## Problem 1 - Lack of expressivity

### Actris 1.0 does not take advantage of the asynchronous semantics

- Bi-directional buffers allows messages in transit in both directions

### Example Program

```
let (c, c') = new_chan () in
fork {send c' 20; let x = recv c' in send c' (x + 2)};
send c 20;
let x = recv c in
let y = recv c in x + y
```

### Dependent separation protocols needed for verification

$$c \mapsto !(x:\mathbb{Z}) \langle x \rangle \{ \text{True} \}. ?\langle 20 \rangle \{ \text{True} \}. ?\langle x + 2 \rangle \{ \text{True} \}. \text{end} \quad \text{and}$$
$$c' \mapsto !\langle 20 \rangle \{ \text{True} \}. ?(x:\mathbb{Z}) \langle x \rangle \{ \text{True} \}. !\langle x + 2 \rangle \{ \text{True} \}. \text{end}$$



## Problem 1 - Lack of expressivity

### Actris 1.0 does not take advantage of the asynchronous semantics

- Bi-directional buffers allows messages in transit in both directions

### Example Program

```
let (c, c') = new_chan () in
fork {send c' 20; let x = recv c' in send c' (x + 2)};
send c 20;
let x = recv c in
let y = recv c in x + y
```

### Dependent separation protocols needed for verification

$$c \mapsto !(x:\mathbb{Z}) \langle x \rangle \{ \text{True} \}. ?\langle 20 \rangle \{ \text{True} \}. ?\langle x + 2 \rangle \{ \text{True} \}. \text{end} \quad \text{and}$$
$$c' \mapsto !\langle 20 \rangle \{ \text{True} \}. ?(x:\mathbb{Z}) \langle x \rangle \{ \text{True} \}. !\langle x + 2 \rangle \{ \text{True} \}. \text{end}$$

### Actris 1.0 requires protocols to be structurally dual

- Every send matched by a receive and vice versa

## Problem 2 - Lack of extensionality

### Example program

```
let (c, c') = new_chan () in  
fork {let w = recv c' in send c' (length w)};  
let v = repeat 42 42 in  
send c v; recv c
```

## Problem 2 - Lack of extensionality

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

Protocols cannot send more than expected

## Problem 2 - Lack of extensionality

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

### Protocols cannot send more than expected

$$c \mapsto !(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ? \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end}$$

## Problem 2 - Lack of extensionality

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

### Protocols cannot send more than expected

$$c \rightsquigarrow !(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ? \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end} \quad \text{and}$$
$$c' \rightsquigarrow ?(v : \text{Val})(\vec{w} : \text{List Val}) \langle v \rangle \{ \text{is\_list } v \vec{w} \}. ! \langle |\vec{w}| \rangle \{ \text{True} \}. \text{end}$$

## Problem 2 - Lack of extensionality

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

### Protocols cannot send more than expected

$$c \rightsquigarrow !(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ? \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end} \quad \text{and}$$
$$c' \rightsquigarrow ?(v : \text{Val})(\vec{w} : \text{List Val}) \langle v \rangle \{ \text{is\_list } v \vec{w} \}. ! \langle |\vec{w}| \rangle \{ \text{True} \}. \text{end}$$

### Actris 1.0 requires protocol payloads to be identical

- One cannot send more or receive less than needed

Integrate **asynchronous session subtyping** with **Actris**

Integrate **asynchronous session subtyping** with **Actris**

**Asynchronous session subtyping** [ [Mostrous et al., Inf.Comput'2015](#) ]

► Swapping:  $?A. !B. S <: !B. ?A. S$



Integrate **asynchronous session subtyping** with **Actris**

**Asynchronous session subtyping** [ [Mostrous et al., Inf.Comput'2015](#) ]

► Swapping:  $?A. !B. S <: !B. ?A. S$

Example:  $?Gift. !Thanks. end <: !Thanks. ?Gift. end$

Integrate **asynchronous session subtyping** with **Actris**

**Asynchronous session subtyping** [ Mostrous et al., Inf.Comput'2015 ]

- ▶ Swapping:  $?A. !B. S <: !B. ?A. S$

Example:  $?Gift. !Thanks. end <: !Thanks. ?Gift. end$

- ▶ Contra and covariance of send / receive: 
$$\frac{B <: A \quad S <: T}{!A. S <: !B. T} \quad \frac{A <: B \quad S <: T}{?A. S <: ?B. T}$$

Integrate **asynchronous session subtyping** with **Actris**

**Asynchronous session subtyping** [ Mostrous et al., Inf.Comput'2015 ]

- ▶ Swapping:  $?A. !B. S <: !B. ?A. S$

Example:  $?Gift. !Thanks. end <: !Thanks. ?Gift. end$

- ▶ Contra and covariance of send / receive: 
$$\frac{B <: A \quad S <: T}{!A. S <: !B. T} \quad \frac{A <: B \quad S <: T}{?A. S <: ?B. T}$$

Example:  $?(\text{List } Z). !Z. end <: ?(\text{List any}). !Z. end$

# Key Idea

Integrate **asynchronous session subtyping** with **Actris**

**Asynchronous session subtyping** [ Mostrous et al., Inf.Comput'2015 ]

- ▶ Swapping:  $?A. !B. S <: !B. ?A. S$

Example:  $?Gift. !Thanks. end <: !Thanks. ?Gift. end$

- ▶ Contra and covariance of send / receive: 
$$\frac{B <: A \quad S <: T}{!A. S <: !B. T} \quad \frac{A <: B \quad S <: T}{?A. S <: ?B. T}$$

Example:  $?(\text{List } Z). !Z. end <: ?(\text{List any}). !Z. end$

- ▶ Subsumption: 
$$\frac{A <: B \quad \Gamma \vdash e : A}{\Gamma \vdash e : B} \quad \frac{S <: T}{\text{chan } S <: \text{chan } T}$$

## Problem 1 - Type Checked

### Example program

```
let (c, c') = new_chan () in
fork {send c' 20; let x = recv c' in send c' (x + 2)};
send c 20;
let x = recv c in
let y = recv c in x + y
```

# Problem 1 - Type Checked

## Example program

```
let (c, c') = new_chan () in
fork {send c' 20; let x = recv c' in send c' (x + 2)};
send c 20;
let x = recv c in
let y = recv c in x + y
```

## Session types needed for type checking

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

# Problem 1 - Type Checked

## Example program

```
let (c, c') = new_chan () in
fork {send c' 20; let x = recv c' in send c' (x + 2)};
send c 20;
let x = recv c in
let y = recv c in x + y
```

## Session types needed for type checking

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

## Allocated dual session types

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

# Problem 1 - Type Checked

## Example program

```
let (c, c') = new_chan () in
fork {send c' 20; let x = recv c' in send c' (x + 2)};
send c 20;
let x = recv c in
let y = recv c in x + y
```

## Session types needed for type checking

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

## Allocated dual session types

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

## Subtype relation of service protocol

$?Z. !Z. !Z. \text{end}$   
 $<: !Z. ?Z. !Z. \text{end}$



## Problem 2 - Type Checked

### Example program

```
let (c, c') = new_chan () in  
fork {let w = recv c' in send c' (length w)};  
let v = repeat 42 42 in  
send c v; recv c
```

## Problem 2 - Type Checked

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

### Corresponding session types

$$\begin{array}{l} c : !(List\ Z).?Z.end \quad \text{and} \\ c' : ?(List\ any).!Z.end \end{array}$$

## Problem 2 - Type Checked

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

### Corresponding session types

$$c : !(List\ Z).?Z.end \quad \text{and} \\ c' : ?(List\ any).!Z.end$$

### Allocated dual session types

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

## Problem 2 - Type Checked

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

### Corresponding session types

$$c : !(List\ Z).?Z.end \quad \text{and} \\ c' : ?(List\ any).!Z.end$$

### Allocated dual session types

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

### Subtype relation of service protocol

$$?(List\ Z).!Z.end \\ <: ?(List\ any).!Z.end$$

# Subprotocols

## Subprotocols

**Swap**  $\frac{?x:\vec{\tau} \langle v \rangle \{P\}. !y:\vec{\sigma} \langle w \rangle \{Q\}. prot}{\sqsubseteq !y:\vec{\sigma} \langle w \rangle \{Q\}. ?x:\vec{\tau} \langle v \rangle \{P\}. prot}$

**Send**  $\frac{\forall \vec{y}:\vec{\sigma}. Q \multimap \exists \vec{x}:\vec{\tau}. P * (v_1 = v_2) * \triangleright (prot_1 \sqsubseteq prot_2)}{!x:\vec{\tau} \langle v_1 \rangle \{P\}. prot_1 \sqsubseteq !y:\vec{\sigma} \langle v_2 \rangle \{Q\}. prot_2}$

**Recv**  $\frac{\forall \vec{x}:\vec{\tau}. P \multimap \exists \vec{y}:\vec{\sigma}. Q * (v_1 = v_2) * \triangleright (prot_1 \sqsubseteq prot_2)}{?x:\vec{\tau} \langle v_1 \rangle \{P\}. prot_1 \sqsubseteq ?y:\vec{\sigma} \langle v_2 \rangle \{Q\}. prot_2}$

**Sub.**  $\frac{prot_1 \sqsubseteq prot_2 \quad c \multimap prot_1}{c \multimap prot_2}$

## Subtyping

$?A. !B. S$   
 $<: !B. ?A. S$

$B <: A \quad S <: T$   
 $!A. S <: !B. T$

$A <: B \quad S <: T$   
 $?A. S <: ?B. T$

$A <: B \quad \Gamma \vdash e : A$   
 $\Gamma \vdash e : B$

# Subprotocols

## Subprotocols

Swap

$$\begin{array}{l} ?\vec{x}:\vec{\tau}\langle v\rangle\{P\}.! \vec{y}:\vec{\sigma}\langle w\rangle\{Q\}. prot \\ \sqsubseteq ! \vec{y}:\vec{\sigma}\langle w\rangle\{Q\}. ?\vec{x}:\vec{\tau}\langle v\rangle\{P\}. prot \end{array}$$

Send

$$\frac{\forall \vec{y}:\vec{\sigma}. Q \multimap \exists \vec{x}:\vec{\tau}. P * (v_1 = v_2) * \triangleright (prot_1 \sqsubseteq prot_2)}{! \vec{x}:\vec{\tau}\langle v_1\rangle\{P\}. prot_1 \sqsubseteq ! \vec{y}:\vec{\sigma}\langle v_2\rangle\{Q\}. prot_2}$$

Recv

$$\frac{\forall \vec{x}:\vec{\tau}. P \multimap \exists \vec{y}:\vec{\sigma}. Q * (v_1 = v_2) * \triangleright (prot_1 \sqsubseteq prot_2)}{? \vec{x}:\vec{\tau}\langle v_1\rangle\{P\}. prot_1 \sqsubseteq ? \vec{y}:\vec{\sigma}\langle v_2\rangle\{Q\}. prot_2}$$

Sub.

$$\frac{prot_1 \sqsubseteq prot_2 \quad c \multimap prot_1}{c \multimap prot_2}$$

## Subtyping

$$\begin{array}{l} ?A. !B. S \\ <: !B. ?A. S \end{array}$$

$$\frac{B <: A \quad S <: T}{!A. S <: !B. T}$$

$$\frac{A <: B \quad S <: T}{?A. S <: ?B. T}$$

$$\frac{A <: B \quad \Gamma \vdash e : A}{\Gamma \vdash e : B}$$

# Subprotocols

## Subprotocols

Swap

$$\begin{array}{l} ?\vec{x}:\vec{\tau} \langle v \rangle \{P\}. !\vec{y}:\vec{\sigma} \langle w \rangle \{Q\}. prot \\ \sqsubseteq !\vec{y}:\vec{\sigma} \langle w \rangle \{Q\}. ?\vec{x}:\vec{\tau} \langle v \rangle \{P\}. prot \end{array}$$

Send

$$\frac{\forall \vec{y}:\vec{\sigma}. Q \multimap \exists \vec{x}:\vec{\tau}. P * (v_1 = v_2) * \triangleright (prot_1 \sqsubseteq prot_2)}{!\vec{x}:\vec{\tau} \langle v_1 \rangle \{P\}. prot_1 \sqsubseteq !\vec{y}:\vec{\sigma} \langle v_2 \rangle \{Q\}. prot_2}$$

Recv

$$\frac{\forall \vec{x}:\vec{\tau}. P \multimap \exists \vec{y}:\vec{\sigma}. Q * (v_1 = v_2) * \triangleright (prot_1 \sqsubseteq prot_2)}{?\vec{x}:\vec{\tau} \langle v_1 \rangle \{P\}. prot_1 \sqsubseteq ?\vec{y}:\vec{\sigma} \langle v_2 \rangle \{Q\}. prot_2}$$

Sub.

$$\frac{prot_1 \sqsubseteq prot_2 \quad c \multimap prot_1}{c \multimap prot_2}$$

## Subtyping

$$\begin{array}{l} ?A. !B. S \\ <: !B. ?A. S \end{array}$$

$$\frac{B <: A \quad S <: T}{!A. S <: !B. T}$$

$$\frac{A <: B \quad S <: T}{?A. S <: ?B. T}$$

$$\frac{A <: B \quad \Gamma \vdash e : A}{\Gamma \vdash e : B}$$

# Subprotocols

## Subprotocols

Swap

$$\begin{array}{l} ?\vec{x}:\vec{\tau} \langle v \rangle \{P\}. !\vec{y}:\vec{\sigma} \langle w \rangle \{Q\}. prot \\ \sqsubseteq !\vec{y}:\vec{\sigma} \langle w \rangle \{Q\}. ?\vec{x}:\vec{\tau} \langle v \rangle \{P\}. prot \end{array}$$

Send

$$\frac{\forall \vec{y}:\vec{\sigma}. Q \multimap \exists \vec{x}:\vec{\tau}. P * (v_1 = v_2) * \triangleright (prot_1 \sqsubseteq prot_2)}{!\vec{x}:\vec{\tau} \langle v_1 \rangle \{P\}. prot_1 \sqsubseteq !\vec{y}:\vec{\sigma} \langle v_2 \rangle \{Q\}. prot_2}$$

Recv

$$\frac{\forall \vec{x}:\vec{\tau}. P \multimap \exists \vec{y}:\vec{\sigma}. Q * (v_1 = v_2) * \triangleright (prot_1 \sqsubseteq prot_2)}{?\vec{x}:\vec{\tau} \langle v_1 \rangle \{P\}. prot_1 \sqsubseteq ?\vec{y}:\vec{\sigma} \langle v_2 \rangle \{Q\}. prot_2}$$

Sub.

$$\frac{prot_1 \sqsubseteq prot_2 \quad c \multimap prot_1}{c \multimap prot_2}$$

## Subtyping

$$\begin{array}{l} ?A. !B. S \\ <: !B. ?A. S \end{array}$$

$$\frac{B <: A \quad S <: T}{!A. S <: !B. T}$$

$$\frac{A <: B \quad S <: T}{?A. S <: ?B. T}$$

$$\frac{A <: B \quad \Gamma \vdash e : A}{\Gamma \vdash e : B}$$



# Subprotocols

## Subprotocols

Swap

$$\begin{array}{l} ?\vec{x}:\vec{\tau} \langle v \rangle \{P\}. !\vec{y}:\vec{\sigma} \langle w \rangle \{Q\}. prot \\ \sqsubseteq !\vec{y}:\vec{\sigma} \langle w \rangle \{Q\}. ?\vec{x}:\vec{\tau} \langle v \rangle \{P\}. prot \end{array}$$

Send

$$\frac{\forall \vec{y}:\vec{\sigma}. Q \multimap \exists \vec{x}:\vec{\tau}. P * (v_1 = v_2) * \triangleright (prot_1 \sqsubseteq prot_2)}{!\vec{x}:\vec{\tau} \langle v_1 \rangle \{P\}. prot_1 \sqsubseteq !\vec{y}:\vec{\sigma} \langle v_2 \rangle \{Q\}. prot_2}$$

Recv

$$\frac{\forall \vec{x}:\vec{\tau}. P \multimap \exists \vec{y}:\vec{\sigma}. Q * (v_1 = v_2) * \triangleright (prot_1 \sqsubseteq prot_2)}{?\vec{x}:\vec{\tau} \langle v_1 \rangle \{P\}. prot_1 \sqsubseteq ?\vec{y}:\vec{\sigma} \langle v_2 \rangle \{Q\}. prot_2}$$

Sub.

$$\frac{prot_1 \sqsubseteq prot_2 \quad c \triangleright prot_1}{c \triangleright prot_2}$$

## Subtyping

$$\begin{array}{l} ?A. !B. S \\ <: !B. ?A. S \end{array}$$

$$\frac{B <: A \quad S <: T}{!A. S <: !B. T}$$

$$\frac{A <: B \quad S <: T}{?A. S <: ?B. T}$$

$$\frac{A <: B \quad \Gamma \vdash e : A}{\Gamma \vdash e : B}$$

## Problem 1 - Verified

### Example program

```
let (c, c') = new_chan () in
fork {send c' 20; let x = recv c' in send c' (x + 2)};
send c 20;
let x = recv c in
let y = recv c in x + y
```

## Problem 1 - Verified

### Example program

```
let (c, c') = new_chan () in
fork {send c' 20; let x = recv c' in send c' (x + 2)};
send c 20;
let x = recv c in
let y = recv c in x + y
```

### Dual dependent separation protocols

$$c \mapsto !(x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ? \langle 20 \rangle \{ \text{True} \}. ? \langle x + 2 \rangle \{ \text{True} \}. \text{end} \quad \text{and}$$
$$c' \mapsto ?(x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ! \langle 20 \rangle \{ \text{True} \}. ! \langle x + 2 \rangle \{ \text{True} \}. \text{end}$$

# Problem 1 - Verified

## Example program

```
let (c, c') = new_chan () in
fork {send c' 20; let x = recv c' in send c' (x + 2)};
send c 20;
let x = recv c in
let y = recv c in x + y
```

## Dual dependent separation protocols

$$c \mapsto !(x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ? \langle 20 \rangle \{ \text{True} \}. ? \langle x + 2 \rangle \{ \text{True} \}. \text{end} \quad \text{and}$$
$$c' \mapsto ?(x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ! \langle 20 \rangle \{ \text{True} \}. ! \langle x + 2 \rangle \{ \text{True} \}. \text{end}$$

## Subprotocol relation of service protocol

$$\begin{aligned} & ?(x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ! \langle 20 \rangle \{ \text{True} \}. ! \langle x + 2 \rangle \{ \text{True} \}. \text{end} \\ \sqsubseteq & ! \langle 20 \rangle \{ \text{True} \}. ?(x : \mathbb{Z}) \langle x \rangle \{ \text{True} \}. ! \langle x + 2 \rangle \{ \text{True} \}. \text{end} \end{aligned}$$

## Problem 2 - Verified

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

## Problem 2 - Verified

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

### Dual dependent separation protocols

$!(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ? \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end}$     and     $?(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ! \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end}$

## Problem 2 - Verified

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

### Dual dependent separation protocols

$$\begin{aligned} &!(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ? \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end} \quad \text{and} \\ &?(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ! \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end} \end{aligned}$$

### Subprotocol relation of service protocol

$$\begin{aligned} &?(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ! \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end} \\ &\sqsubseteq ?(v : \text{Val})(\vec{w} : \text{List Val}) \langle v \rangle \{ \text{is\_list } v \vec{w} \}. ! \langle |\vec{w}| \rangle \{ \text{True} \}. \text{end} \end{aligned}$$

## Problem 2 - Verified

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

### Dual dependent separation protocols

$$\begin{aligned} &!(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ? \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end} \quad \text{and} \\ &?(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ! \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end} \end{aligned}$$

### Subprotocol relation of service protocol

$$\begin{aligned} &?(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ! \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end} \\ \sqsubseteq &?(v : \text{Val})(\vec{w} : \text{List Val}) \langle v \rangle \{ \text{is\_list } v \vec{w} \}. ! \langle |\vec{w}| \rangle \{ \text{True} \}. \text{end} \\ &\forall (v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}). \text{is\_int\_list } v \vec{x} \rightarrow * \end{aligned}$$



## Problem 2 - Verified

### Example program

```
let (c, c') = new_chan () in
fork {let w = recv c' in send c' (length w)};
let v = repeat 42 42 in
send c v; recv c
```

### Dual dependent separation protocols

$$\begin{aligned} &!(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ? \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end} \quad \text{and} \\ &?(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ! \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end} \end{aligned}$$

### Subprotocol relation of service protocol

$$\begin{aligned} &?(v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}) \langle v \rangle \{ \text{is\_int\_list } v \vec{x} \}. ! \langle |\vec{x}| \rangle \{ \text{True} \}. \text{end} \\ \sqsubseteq &?(v : \text{Val})(\vec{w} : \text{List Val}) \langle v \rangle \{ \text{is\_list } v \vec{w} \}. ! \langle |\vec{w}| \rangle \{ \text{True} \}. \text{end} \\ &\forall (v : \text{Val})(\vec{x} : \text{List } \mathbb{Z}). \text{is\_int\_list } v \vec{x} \rightarrow \\ &\exists (v : \text{Val})(\vec{w} : \text{List Val}). \text{is\_list } v \vec{w} * \dots \end{aligned}$$

## Additional properties of subprotocols

**Expressivity beyond asynchronous session subtyping**

## Additional properties of subprotocols

**Expressivity beyond asynchronous session subtyping**

**Eagerly resolving obligations**

## Additional properties of subprotocols

**Expressivity beyond asynchronous session subtyping**

**Eagerly resolving obligations**

$$\frac{P}{!\langle v \rangle \{P * Q\}. \text{end} \sqsubseteq !\langle v \rangle \{Q\}. \text{end}}$$

# Additional properties of subprotocols

**Expressivity beyond asynchronous session subtyping**

**Eagerly resolving obligations**

$$\frac{P}{!\langle v \rangle \{P * Q\}. \text{end} \sqsubseteq !\langle v \rangle \{Q\}. \text{end}}$$

**Sending and recovering a “frame”**

$$\frac{\{P\} e \{Q\}}{\{P * R\} e \{Q * R\}}$$

# Additional properties of subprotocols

**Expressivity beyond asynchronous session subtyping**

**Eagerly resolving obligations**

$$\frac{P}{!\langle v \rangle \{P * Q\}. \text{end} \sqsubseteq !\langle v \rangle \{Q\}. \text{end}}$$

**Sending and recovering a “frame”**  $\frac{\{P\} e \{Q\}}{\{P * R\} e \{Q * R\}}$

$$\begin{aligned} & !\langle v \rangle \{P\}. ?\langle w \rangle \{Q\}. \text{end} \\ \sqsubseteq & !\langle v \rangle \{P * R\}. ?\langle w \rangle \{Q * R\}. \text{end} \end{aligned}$$

# Additional properties of subprotocols

**Expressivity beyond asynchronous session subtyping**

**Eagerly resolving obligations**

$$\frac{P}{!\langle v \rangle \{P * Q\}. \text{end} \sqsubseteq !\langle v \rangle \{Q\}. \text{end}}$$

**Sending and recovering a “frame”**  $\frac{\{P\} e \{Q\}}{\{P * R\} e \{Q * R\}}$

$$\begin{aligned} & !\langle v \rangle \{P\}. ?\langle w \rangle \{Q\}. \text{end} \\ \sqsubseteq & !\langle v \rangle \{P * R\}. ?\langle w \rangle \{Q * R\}. \text{end} \end{aligned}$$

**Löb-based reasoning for non-structural subprotocol relations**

# Additional properties of subprotocols

**Expressivity beyond asynchronous session subtyping**

**Eagerly resolving obligations**

$$\frac{P}{! \langle v \rangle \{ P * Q \}. \text{end} \sqsubseteq ! \langle v \rangle \{ Q \}. \text{end}}$$

**Sending and recovering a “frame”**  $\frac{\{P\} e \{Q\}}{\{P * R\} e \{Q * R\}}$

$$\begin{aligned} & ! \langle v \rangle \{ P \}. ? \langle w \rangle \{ Q \}. \text{end} \\ \sqsubseteq & ! \langle v \rangle \{ P * R \}. ? \langle w \rangle \{ Q * R \}. \text{end} \end{aligned}$$

**Löb-based reasoning for non-structural subprotocol relations**

$$\begin{aligned} & \mu \text{rec}. ! \langle 42 \rangle \{ \text{True} \}. \text{rec} \\ \sqsubseteq & \mu \text{rec}. ! \langle 42 \rangle \{ \text{True} \}. ! \langle 42 \rangle \{ \text{True} \}. \text{rec} \end{aligned}$$



Draft [LMCS]: [https://itu.dk/people/jkas/papers/actris\\_lmcs.pdf](https://itu.dk/people/jkas/papers/actris_lmcs.pdf)  
Mechanisation: <https://gitlab.mpi-sws.org/iris/actris/-/tree/lmcs>

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

JONAS KASTBERG HINRICHSSEN, JESPER BENGTON, AND ROBBERT KREBBERS

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

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

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

**ABSTRACT.** 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 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

# Soundness and implementation of Actris

If  $\{\text{True}\} e \{v. \phi(v)\}$  is provable in Actris then:

- ✓ **Safety/session fidelity:**  $e$  will not crash and not send wrong messages
- ✓ **Functional correctness:** If  $e$  terminates with  $v$ , the postcondition  $\phi(v)$  holds

# Implementation and model of Actris in Iris

## **Approach:**

- ▶ Define the type of *prot* with support from Iris's recursive domain equation solver

# Implementation and model of Actris in Iris

## Approach:

- ▶ Define the type of *prot* with support from Iris's recursive domain equation solver
- ▶ Define operations and relations on *prot*, such as  $\overline{prot}$  and  $prot_1 \sqsubseteq prot_2$

# Implementation and model of Actris in Iris

## Approach:

- ▶ Define the type of *prot* with support from Iris's recursive domain equation solver
- ▶ Define operations and relations on *prot*, such as  $\overline{\text{prot}}$  and  $\text{prot}_1 \sqsubseteq \text{prot}_2$
- ▶ Prove the rules of a *ghost theory* API, connecting *prot* to intuitive resources

# Implementation and model of Actris in Iris

## Approach:

- ▶ Define the type of *prot* with support from Iris's recursive domain equation solver
- ▶ Define operations and relations on *prot*, such as  $\overline{prot}$  and  $prot_1 \sqsubseteq prot_2$
- ▶ Prove the rules of a *ghost theory* API, connecting *prot* to intuitive resources
- ▶ Implement `new_chan`, `send`, and `recv` as a library using lock-protected buffers

# Implementation and model of Actris in Iris

## Approach:

- ▶ Define the type of *prot* with support from Iris's recursive domain equation solver
- ▶ Define operations and relations on *prot*, such as  $\overline{\text{prot}}$  and  $\text{prot}_1 \sqsubseteq \text{prot}_2$
- ▶ Prove the rules of a *ghost theory* API, connecting *prot* to intuitive resources
- ▶ Implement `new_chan`, `send`, and `recv` as a library using lock-protected buffers
- ▶ Define  $c \rightsquigarrow \text{prot}$  using Iris's invariants and the ghost theory



# Implementation and model of Actris in Iris

## Approach:

- ▶ Define the type of  $prot$  with support from Iris's recursive domain equation solver
- ▶ Define operations and relations on  $prot$ , such as  $\overline{prot}$  and  $prot_1 \sqsubseteq prot_2$
- ▶ Prove the rules of a *ghost theory* API, connecting  $prot$  to intuitive resources
- ▶ Implement `new_chan`, `send`, and `recv` as a library using lock-protected buffers
- ▶ Define  $c \rightsquigarrow prot$  using Iris's invariants and the ghost theory
- ▶ Prove Actris's proof rules as lemmas in Iris

# Implementation and model of Actris in Iris

## Approach:

- ▶ Define the type of *prot* with support from Iris's recursive domain equation solver
- ▶ Define operations and relations on *prot*, such as  $\overline{prot}$  and  $prot_1 \sqsubseteq prot_2$
- ▶ Prove the rules of a *ghost theory* API, connecting *prot* to intuitive resources
- ▶ Implement `new_chan`, `send`, and `recv` as a library using lock-protected buffers
- ▶ Define  $c \rightsquigarrow prot$  using Iris's invariants and the ghost theory
- ▶ Prove Actris's proof rules as lemmas in Iris

## Benefits:

- ✓ Actris's soundness result is a corollary of Iris's soundness

# Implementation and model of Actris in Iris

## Approach:

- ▶ Define the type of *prot* with support from Iris's recursive domain equation solver
- ▶ Define operations and relations on *prot*, such as  $\overline{\text{prot}}$  and  $\text{prot}_1 \sqsubseteq \text{prot}_2$
- ▶ Prove the rules of a *ghost theory* API, connecting *prot* to intuitive resources
- ▶ Implement `new_chan`, `send`, and `recv` as a library using lock-protected buffers
- ▶ Define  $c \rightsquigarrow \text{prot}$  using Iris's invariants and the ghost theory
- ▶ Prove Actris's proof rules as lemmas in Iris

## Benefits:

- ✓ Actris's soundness result is a corollary of Iris's soundness
- ✓ Readily integrates with other concurrency mechanisms in Iris

# Implementation and model of Actris in Iris

## Approach:

- ▶ Define the type of *prot* with support from Iris's recursive domain equation solver
- ▶ Define operations and relations on *prot*, such as  $\overline{\text{prot}}$  and  $\text{prot}_1 \sqsubseteq \text{prot}_2$
- ▶ Prove the rules of a *ghost theory* API, connecting *prot* to intuitive resources
- ▶ Implement `new_chan`, `send`, and `recv` as a library using lock-protected buffers
- ▶ Define  $c \rightsquigarrow \text{prot}$  using Iris's invariants and the ghost theory
- ▶ Prove Actris's proof rules as lemmas in Iris

## Benefits:

- ✓ Actris's soundness result is a corollary of Iris's soundness
- ✓ Readily integrates with other concurrency mechanisms in Iris
- ✓ Can readily reuse Iris's support for interactive proofs in Coq

# Implementation and model of Actris in Iris

## Approach:

- ▶ Define the type of *prot* with support from Iris's recursive domain equation solver
- ▶ Define operations and relations on *prot*, such as  $\overline{\text{prot}}$  and  $\text{prot}_1 \sqsubseteq \text{prot}_2$
- ▶ Prove the rules of a *ghost theory* API, connecting *prot* to intuitive resources
- ▶ Implement `new_chan`, `send`, and `recv` as a library using lock-protected buffers
- ▶ Define  $c \rightsquigarrow \text{prot}$  using Iris's invariants and the ghost theory
- ▶ Prove Actris's proof rules as lemmas in Iris

## Benefits:

- ✓ Actris's soundness result is a corollary of Iris's soundness
- ✓ Readily integrates with other concurrency mechanisms in Iris
- ✓ Can readily reuse Iris's support for interactive proofs in Coq
- ✓ Small Coq development ( $\sim 5000$  lines in total)

Draft [LMCS]: [https://itu.dk/people/jkas/papers/actris\\_lmcs.pdf](https://itu.dk/people/jkas/papers/actris_lmcs.pdf)  
Mechanisation: <https://gitlab.mpi-sws.org/iris/actris/-/tree/lmcs>

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

JONAS KASTBERG HINRICHSSEN, JESPER BENGTON, AND ROBBERT KREBBERS

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

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

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

---

**ABSTRACT.** 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 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

# Semantic Session Typing

joint work with

Daniël Louwrik, University of Amsterdam

Jesper Bengtson, IT University of Copenhagen

Robbert Krebbers, Radboud University

**No formal connection between dependent separation protocols and session types**

- ▶ Protocols merely designed in the style of session types



## **No formal connection between dependent separation protocols and session types**

- ▶ Protocols merely designed in the style of session types

## **Lack of expressivity of existing session type systems**

- ▶ Polymorphism, recursion, and subtyping have been studied individually
- ▶ No session type system that combines all three

## **No formal connection between dependent separation protocols and session types**

- ▶ Protocols merely designed in the style of session types

## **Lack of expressivity of existing session type systems**

- ▶ Polymorphism, recursion, and subtyping have been studied individually
- ▶ No session type system that combines all three

## **Lack of mechanised soundness proofs for session type systems**

- ▶ Few results exist for simpler systems
- ▶ None exist for more expressive systems

## Semantic Typing

**Semantic Typing** [Milner, Princeton Proof-Carrying Code project, RustBelt Project]

- Types defined as predicates over values:  $Z \triangleq \lambda w. w \in \mathbb{Z}$

## Semantic Typing

**Semantic Typing** [Milner, Princeton Proof-Carrying Code project, RustBelt Project]

- ▶ Types defined as predicates over values:  $Z \triangleq \lambda w. w \in \mathbb{Z}$
- ▶ Judgement defined as safety-capturing evaluation:  $\Gamma \models e : A$

## Semantic Typing

**Semantic Typing** [Milner, Princeton Proof-Carrying Code project, RustBelt Project]

- ▶ Types defined as predicates over values:  $Z \triangleq \lambda w. w \in \mathbb{Z}$
- ▶ Judgement defined as safety-capturing evaluation:  $\Gamma \models e : A$
- ▶ Rules proven as lemmas:  $\models i : Z$

## Semantic Typing

**Semantic Typing** [Milner, Princeton Proof-Carrying Code project, RustBelt Project]

- ▶ Types defined as predicates over values:  $Z \triangleq \lambda w. w \in \mathbb{Z}$
- ▶ Judgement defined as safety-capturing evaluation:  $\Gamma \models e : A$
- ▶ Rules proven as lemmas:  $\vdash i : Z \rightsquigarrow i \in \mathbb{Z}$

## Semantic Typing

**Semantic Typing** [Milner, Princeton Proof-Carrying Code project, RustBelt Project]

- ▶ Types defined as predicates over values:  $Z \triangleq \lambda w. w \in \mathbb{Z}$
- ▶ Judgement defined as safety-capturing evaluation:  $\Gamma \models e : A$
- ▶ Rules proven as lemmas:  $\vdash i : Z \rightsquigarrow i \in \mathbb{Z}$
- ▶ Soundness inherited from underlying logic

## Semantic Typing using Iris

**Semantic Typing** [Milner, Princeton Proof-Carrying Code project, RustBelt Project]

- ▶ Types defined as predicates over values:  $Z \triangleq \lambda w. w \in \mathbb{Z}$
- ▶ Judgement defined as safety-capturing evaluation:  $\Gamma \models e : A$
- ▶ Rules proven as lemmas:  $\vdash i : Z \rightsquigarrow i \in \mathbb{Z}$
- ▶ Soundness inherited from underlying logic

**Iris** [Iris project]

- ▶ Semantic type system for ML-like language with concurrency and state  
<https://gitlab.mpi-sws.org/iris/tutorial-popl20>
- ▶ Mechanised in **Coq**



## Semantic Typing using Iris and Actris

**Semantic Typing** [Milner, Princeton Proof-Carrying Code project, RustBelt Project]

- ▶ Types defined as predicates over values:  $Z \triangleq \lambda w. w \in \mathbb{Z}$
- ▶ Judgement defined as safety-capturing evaluation:  $\Gamma \models e : A$
- ▶ Rules proven as lemmas:  $\vdash i : Z \rightsquigarrow i \in \mathbb{Z}$
- ▶ Soundness inherited from underlying logic

**Iris** [Iris project]

- ▶ Semantic type system for ML-like language with concurrency and state  
<https://gitlab.mpi-sws.org/iris/tutorial-popl20>
- ▶ Mechanised in **Coq**

**Actris** [Hinrichsen et al., POPL'20]

- ▶ **Dependent separation protocols:** Session type-style logical protocols
- ▶ Mechanised in **Coq**

## Semantic Session Type System

- ▶ Formal connection between dependent separation protocols and session types
- ▶ Rich extensible type system for session types
  - ▶ Term and session type equi-recursion
  - ▶ Term and session type polymorphism
  - ▶ Term and (asynchronous) session type subtyping
  - ▶ Unique and shared reference types, copyable types, lock types
- ▶ Full mechanisation in Coq
- ▶ Supports integration of safe yet untypeable programs

Paper [CPP'21]: [https://itu.dk/people/jkas/papers/semantic\\_session\\_typing\\_cpp.pdf](https://itu.dk/people/jkas/papers/semantic_session_typing_cpp.pdf)  
Mechanisation: <https://gitlab.mpi-sws.org/iris/actris/-/tree/cpp21>

## Machine-Checked Semantic Session Typing

Jonas Kastberg Hinrichsen  
IT University of Copenhagen, Denmark

Robbert Krebbers  
Radboud University and Delft University of Technology,  
The Netherlands

Daniël Louwrik  
University of Amsterdam, 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 the typing judgements of racy, but safe, programs that cannot be type checked using only the rules of the type system.

using *logical relations* defined in terms of a program logic [Appel et al. 2007; Dreyer et al. 2009, 2019].

The semantic approach addresses the challenges above as (1) typing judgements are definitions in the program logic, and typing rules are lemmas in the program logic (they are not inductively defined), which means that extending the system with new typing rules boils down to proving the corresponding typing lemmas correct; (2) safe functions that cannot be conventionally type checked can still be semantically type checked by manually proving a typing lemma (3) all of our results have been mechanised in Coq using Iris [Jung et al. 2016, 2018b, 2015; Krebbers et al. 2018, 2017a,b] giving us a high degree of trust that they are correct.

The syntactic approach requires global proofs of progress (well-typed programs are either values or can take a step) and preservation (steps taken by the program do not change types), culminating in type safety (well-typed programs do

# Ongoing and future work

# Ongoing and future work

## Ongoing work

- ▶ Dependent separation protocols as specifications for TCP-based communication in distributed systems
- ▶ Multi-party dependent separation protocols (based on [ [Honda et al., POPL'08](#) ])

## Future Work

- ▶ Deadlock free communication (based on ongoing work by [Jules Jacobs](#))
- ▶ Linearity of channels through Iron [ [Bizjak et al., POPL'19](#) ]

!  $\langle \text{"Thank you"} \rangle \{ \text{ActrisKnowledge} \}.$   
 $\mu rec. ?(q : \text{Question}) \langle q \rangle \{ \text{AboutActris } q \}.$   
    !  $(a : \text{Answer}) \langle a \rangle \{ \text{Insightful } q \ a \}. rec$