Types and Contracts for Binary Sessions

from **theory** to **practice**

Luca Padovani

Dipartimento di Informatica, Università di Torino

Introduction to binary sessions

binary sessions in a nutshell

- **private** communication channel between two processes
- each **endpoint** has a **session type** (= protocol description)
- **peer** endpoints have **dual** session types

some properties and methods to enforce them

property	counterexample		method	at
protocol fidelity	send recv	send recv	session types	compile time*
comm. safety	send int	recv bool	session types	compile time*
blame correctness	send 0	recv ($\neq 0$)	contracts	runtime

T, S	$::=$	end	end of conversation
	$\mid$	$!t\,.\,T$	send message of type t
	$\mid$	$?t\,.\,T$	receive message of type t
	$\mid$	$T \oplus S$	choose T or S
	$\mid$	$T \,\&\, S$	offer T and S

?int.?int.!int

example

```
let client a =                             a : !int.!int.?int
  let a = send 123 a in                    a : !int.?int
  let a = send 45 a in                     a : ?int
  let r, a = recv a in ...                 a : end
```

example

```
let client a =                          a : !int.!int.?int
  let a = send 123 a in                 a : !int.?int
  let a = send 45 a in                  a : ?int
  let r, a = recv a in ...              a : end

let server b =                          b : ?int.?int.!int
  let x, b = recv b in                  b : ?int.!int
  let y, b = recv b in                  b : !int
  let b = send (x mod y) b in ...       b : end
```

example

```
let client a =                         a : !int.!int.?int
  let a = send 123 a in                a : !int.?int
  let a = send 45 a in                 a : ?int
  let r, a = recv a in ...             a : end

let server b =                         b : ?int.?int.!int
  let x, b = recv b in                 b : ?int.!int
  let y, b = recv b in                 b : !int
  let b = send (x mod y) b in ...      b : end

let main () =
  let a, b = open () in           (* a b ⇒ dual types *)
  spawn server b;
  spawn client a
```

endpoints are linear resources

```
let client a =                        a : !int.!int.?int
  let _ = send 123 a in               a : !int.!int.?int
  let a = send 234 a in               a : !int.!int ☠
  let a = send 45 a in                a : ?int
  let r, a = recv a in ...            a : end
```

- the “same” endpoint cannot be used more than once

⇒ **substructural** type system

session API

$$\begin{aligned} \texttt{open} &: \text{unit} \rightarrow T \times \overline{T} \qquad \textbf{duality} \\ \texttt{send} &: t \rightarrow \texttt{!}t\,.\,T \rightarrow T \\ \texttt{recv} &: \texttt{?}t\,.\,T \rightarrow t \times T \end{aligned}$$

+ endpoint **linearity**

Theorem (soundness)

Well-typed *programs satisfy* protocol fidelity & communication safety.

Sessions for real

let's play a game

Implement the following interaction with one-shot channels

$$c\,!\,[123].c\,!\,[45].c?(r) \qquad c?(x).c?(y).c\,!\,[x \,\%\, y]$$

let's play a game

Implement the following interaction with one-shot channels

$$c\,!\,[123].c\,!\,[45].c?(r) \qquad c?(x).c?(y).c\,!\,[x \,\%\, y]$$

Sessions in continuation-passing style

$c\,!\,[123, c']$

- message = payload + continuation

let's play a game

Implement the following interaction with one-shot channels

$$c\,!\,[123].c\,!\,[45].c?(r) \qquad c?(x).c?(y).c\,!\,[x \,\%\, y]$$

Sessions in continuation-passing style

$c\,!\,[123, c']$ $\qquad$ $c?(x, a)$

- message = payload + continuation

Implement the following interaction with one-shot channels

$$c\,!\,[123].c\,!\,[45].c?(r) \qquad c?(x).c?(y).c\,!\,[x \,\%\, y]$$

Sessions in continuation-passing style

$c\,!\,[123, c'].c'\,!\,[45, c'']$ $\qquad$ $c?(x, a)$

- message = payload + continuation

let's play a game

Implement the following interaction with one-shot channels

$$c\,!\,[123].c\,!\,[45].c?(r) \qquad c?(x).c?(y).c\,!\,[x \,\%\, y]$$

Sessions in continuation-passing style

$c\,!\,[123, c'].c'\,!\,[45, c'']$ $\qquad$ $c?(x, a).a?(y, b)$

- message = payload + continuation

let's play a game

Implement the following interaction with one-shot channels

$$c\,!\,[123].c\,!\,[45].c?(r) \qquad c?(x).c?(y).c\,!\,[x \,\%\, y]$$

Sessions in continuation-passing style

$$c\,!\,[123, c'].c'\,!\,[45, c''].c''?(r, d) \qquad c?(x, a).a?(y, b)$$

- message = payload + continuation

let's play a game

Implement the following interaction with one-shot channels

$$c\,!\,[123].c\,!\,[45].c?(r) \qquad c?(x).c?(y).c\,!\,[x \mathbin{\%} y]$$

Sessions in continuation-passing style

$$c\,!\,[123, c'].c'\,!\,[45, c''].c''?(r, d) \qquad c?(x, a).a?(y, b).b\,!\,[x \mathbin{\%} y, c''']$$

- message = payload + continuation

binary sessions can be encoded into the linear π-calculus

Relevant literature

- Kobayashi, Pierce, and Turner [1999]
- Kobayashi [2002]
- Demangeon and Honda [2011]
- Dardha, Giachino, and Sangiorgi [2017]

Lifted features and properties

- communication safety
- race freedom
- subtyping for session types
- ...

a welcome side effect on types: duality simplifies!

$\langle t, s \rangle$ = type of a one-shot channel for receiving t or sending s

$c\,!\,[123, c'].c'\,!\,[45, c''].c''?(r, d)$ $\qquad$ $\underbrace{c?(x, a)}.a?(y, b).b\,!\,[x \,\%\, y, c''']$

$\langle \mathsf{int} \times \qquad\qquad , \bullet \rangle$

a welcome side effect on types: duality simplifies!

$\langle t, s \rangle$ = type of a one-shot channel for receiving t or sending s

$c\,!\,[123, c'].c'\,!\,[45, c''].c''?(r, d) \qquad c?(x, a).\underbrace{a?(y, b)}.b\,!\,[x \,\%\, y, c''']$

$\langle \mathsf{int} \times \langle \mathsf{int} \times \qquad , \bullet \rangle, \bullet \rangle$

a welcome side effect on types: duality simplifies!

$\langle t, s \rangle$ = type of a one-shot channel for receiving t or sending s

$c\,!\,[123, c'].c'\,!\,[45, c''].c''?(r, d) \qquad c?(x, a).a?(y, b).\underbrace{b\,!\,[x \mathbin{\%} y, c''']}$

$\langle \mathsf{int} \times \langle \mathsf{int} \times \langle \bullet, \mathsf{int} \times \qquad \rangle, \bullet \rangle, \bullet \rangle$

a welcome side effect on types: duality simplifies!

$\langle t, s \rangle$ = type of a one-shot channel for receiving t or sending s

$$c\,!\,[123, c'].c'\,!\,[45, c''].c''?(r, d) \qquad c?(x, a).a?(y, b).b\,!\,[x \,\%\, y, c''']$$

$$\langle \mathsf{int} \times \langle \mathsf{int} \times \langle \bullet, \mathsf{int} \times \langle \bullet, \bullet \rangle \rangle, \bullet \rangle, \bullet \rangle$$

a welcome side effect on types: duality simplifies!

$\langle t, s\rangle$ = type of a one-shot channel for receiving t or sending s

$\underbrace{c\,!\,[123, c']}.c'\,!\,[45, c''].c''?(r, d) \qquad c?(x, a).a?(y, b).b\,!\,[x \,\%\, y, c''']$

$\langle \bullet, \mathsf{int} \times \qquad \rangle \quad \langle \mathsf{int} \times \langle \mathsf{int} \times \langle \bullet, \mathsf{int} \times \langle \bullet, \bullet\rangle\rangle, \bullet\rangle, \bullet\rangle$

a welcome side effect on types: duality simplifies!

$\langle t, s \rangle$ = type of a one-shot channel for receiving t or sending s

$c\,!\,[123, c'].c'\,!\,[45, c''].c''?(r, d)$ $\qquad$ $c?(x, a).\underbrace{a?(y, b)}.b\,!\,[x \,\%\, y, c''']$

$\langle \bullet, \mathsf{int} \times \langle \mathsf{int} \times \qquad , \bullet \rangle \rangle$ $\quad$ $\langle \mathsf{int} \times \langle \mathsf{int} \times \langle \bullet, \mathsf{int} \times \langle \bullet, \bullet \rangle \rangle, \bullet \rangle, \bullet \rangle$

a welcome side effect on types: duality simplifies!

$\langle t, s \rangle$ = type of a one-shot channel for receiving t or sending s

$c\,!\,[123, c'].c'\,!\,[45, c''].c''?(r, d)$ $\qquad$ $c?(x, a).a?(y, b).\underbrace{b\,!\,[x \,\%\, y, c''']}$

$\langle \bullet, \mathsf{int} \times \langle \mathsf{int} \times \langle \bullet, \mathsf{int} \times \langle \bullet, \bullet \rangle\rangle, \bullet \rangle\rangle$ $\langle \mathsf{int} \times \langle \mathsf{int} \times \langle \bullet, \mathsf{int} \times \langle \bullet, \bullet \rangle\rangle, \bullet \rangle, \bullet \rangle$

a welcome side effect on types: duality simplifies!

$\langle t, s \rangle$ = type of a one-shot channel for receiving t or sending s

$c\,!\,[123, c'].c'\,!\,[45, c''].c''?(r, d) \qquad c?(x, a).a?(y, b).b\,!\,[x \,\%\, y, c''']$

$\langle \bullet, \mathsf{int} \times \langle \mathsf{int} \times \langle \bullet, \mathsf{int} \times \langle \bullet, \bullet \rangle\rangle, \bullet \rangle\rangle \; \langle \mathsf{int} \times \langle \mathsf{int} \times \langle \bullet, \mathsf{int} \times \langle \bullet, \bullet \rangle\rangle, \bullet \rangle, \bullet \rangle$

$$\langle \bullet, \mathsf{int} \times \langle \mathsf{int} \times \langle \bullet, \mathsf{int} \times \langle \bullet, \bullet \rangle\rangle, \bullet \rangle\rangle$$
$$\langle \mathsf{int} \times \langle \mathsf{int} \times \langle \bullet, \mathsf{int} \times \langle \bullet, \bullet \rangle\rangle, \bullet \rangle, \bullet \rangle$$

Proposition (duality as equality)

If $T \rightsquigarrow \langle t, s \rangle$*, then* $\overline{T} \rightsquigarrow \langle s, t \rangle$

Things we get for free

- **duality**

Proposition (duality as equality)

If $T \rightsquigarrow \langle t, s \rangle$*, then* $\overline{T} \rightsquigarrow \langle s, t \rangle$

Things we get for free

- **duality**
- session type **inference** (lots of previous attempts!)

session API, encoded [Padovani, 2017b]

- represent session types in encoded form...
- ...as if continuations were exchanged...
- ...but don't exchange continuations

session API

$$\begin{array}{lcllcl} \texttt{open} & : & \mathsf{unit} \to T \times \overline{T} & \rightsquigarrow & \mathsf{unit} \to \langle \alpha, \beta \rangle \times \langle \beta, \alpha \rangle \\ \texttt{send} & : & t \to\, !t\,.\,T \to T & \rightsquigarrow & t \to \langle \bullet, t \times \langle \alpha, \beta \rangle \rangle \to \langle \beta, \alpha \rangle \\ \texttt{recv} & : & ?t\,.\,T \to t \times T & \rightsquigarrow & \langle t \times \langle \alpha, \beta \rangle, \bullet \rangle \to t \times \langle \alpha, \beta \rangle \end{array}$$

the ostrich approach to linearity

- **ignore** linearity at the type level
- detect linearity violations **at runtime** (easy and cheap!)
- many linearity violations are statically detected **anyway**

runtime detection of linearity violations

Strategy

- endpoint a^p = **pair** with channel a and **flag** p
- $a^{\texttt{tt}}$ is used $\Rightarrow$ **reset** flag imperatively and **regenerate** pair
- $a^{\texttt{ff}}$ is used $\Rightarrow$ raise **exception**

Proposition

A linearity exception is raised as soon as (but not before) a linearity violation occurs

Observation

Actual measurements indicate that the overhead of runtime linearity violation detection is negligible [Padovani, 2017b]

Context-free session types

modeling a non-uniform object using sessions

```
let stack =
  let rec empty c =
    match branch c with
    | Push c → let x, c = recv c in
               empty (non_empty x c)
    | Stop c → c
  and non_empty x c = (* x on top *)
    match branch c with
    | Push c → let y, c = recv c in
               non_empty x (non_empty y c)
    | Pop c → send x c
  in empty
```

modeling a non-uniform object using sessions

```
let stack =
  let rec empty c =
    match branch c with
    | Push c → let x, c = recv c in
               empty (non_empty x c)
    | Stop c → c                          ☠ dead code
  and non_empty x c = (* x on top *)
    match branch c with
    | Push c → let y, c = recv c in
               non_empty x (non_empty y c)
    | Pop c → send x c                    ☠ dead code
  in empty
```

from ordinary to context-free session types

Ordinary session types

- sequential composition limited to prefixes $?\alpha\,.\,S$
- language of (finite) traces is **regular**

Context-free session types **[Thiemann and Vasconcelos, 2016]**

- general form of sequential composition $T\,.\,S$
- language of (finite) traces is **context-free**
- typability++, precision++

Thiemann and Vasconcelos's type system

Key ingredients

- monoidal laws for sequential composition, *e.g.*

$$\frac{\Gamma \vdash e : T\,.\,(S\,.\,R)}{\Gamma \vdash e : (T\,.\,S)\,.\,R}$$

- polymorphic recursion

Observation

- type inference is **undecidable**
- type checking is arguably **more difficult** (open problem)

resuming endpoints [Padovani, 2017a, 2019]

If $f : T \rightarrow$ end, then

- $(f\ u)$ carries out protocol T on u, and
- returns the **expired** endpoint $u : \mathsf{end}$

If $f : T \rightarrow$ end, then

- $(f\ u)$ carries out protocol T on u, and
- returns the **expired** endpoint $u : \text{end}$

But then $f : T . S \rightarrow S$, meaning that

- $(f\ u)$ carries out protocol T on u, and
- returns the endpoint $u : S$

If $f : T \rightarrow \mathsf{end}$, then

- $(f\ u)$ carries out protocol T on u, and
- returns the **expired** endpoint $u : \mathsf{end}$

But then $f : T \,.\, S \rightarrow S$, meaning that

- $(f\ u)$ carries out protocol T on u, and
- returns the endpoint $u : S$

Idea

- **coerce** $f : T \rightarrow \mathsf{end} \Rightarrow T \,.\, S \rightarrow S$
- ask programmer to place coercions `@¿`

If $f : T \to \mathsf{end}$, then

- $(f\ u)$ carries out protocol T on u, and
- returns ~~the~~ an **expired** endpoint $u : \mathsf{end}$

But then $f : T \,.\, S \to S$, meaning that

- $(f\ u)$ carries out protocol T on u, and
- returns the endpoint $u : S$

Idea

- **coerce** $f : T \to \mathsf{end} \Rightarrow T \,.\, S \to S$
- ask programmer to place coercions @¿

session types with endpoint identities[Padovani, 2017a, 2019]

$$[T]_\varrho$$

session API with endpoint identities

$$\begin{array}{rcl} \texttt{open} & : & \mathsf{unit} \to \exists \varrho, \sigma.([T]_\varrho \times [\overline{T}]_\sigma) \\ \texttt{send} & : & t \to [!t.T]_\varrho \to [T]_\varrho \\ \texttt{recv} & : & [?t.T]_\varrho \to t \times [T]_\varrho \\ \texttt{@¿} & : & ([T]_\varrho \to [\mathsf{end}]_\varrho) \to [T.S]_\varrho \to [S]_\varrho \end{array}$$

Theorem (soundness)

Well-typed programs (with coercions) *satisfy...*

the stack with coercions

```
let stack =
  let rec empty c =
    match branch c with
    | Push c → let x, c = recv u in
               empty (non_empty x @> c)
    | Stop c → c
  and non_empty x c  =
    match branch c with
    | Push c → let y, c = recv c in
               non_empty x (non_empty y @> c)
    | Pop c → send x c
  in empty
```

Chaperone contracts for sessions

`!int.!int.?int`

1. send a number
2. send a number
3. receive a number

from types to contracts [Melgratti and Padovani, 2017]

$$!\mathtt{int}.!\mathtt{int}.?\mathtt{int}$$

1. send a number
2. send a number $\neq 0$
3. receive a number ≥ 0

$$!*.!(\neq 0).?(\geq 0)$$

- **monitor** sessions at runtime
- **blame** guilty process when a contract violation is detected

a DSL for contracts

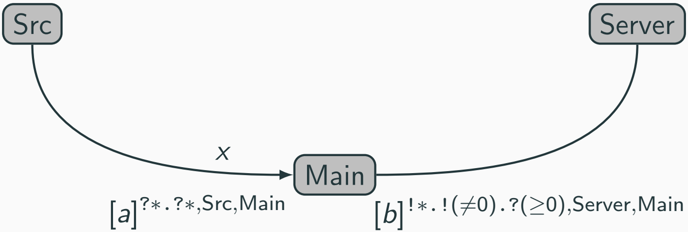
```
let server b = ... (* as before *)

let contract =
  send_c any_c @@
  send_c (flat_c (≠ 0)) @@
  recv_c (flat_c (≥ 0)) @@
  end_c

let server_chan = register server contract "Server"

let main () =
  let b = connect server_chan "Main" in ...
```

monitored session endpoints

$$[u]^{\mathrm{C},p,q}$$

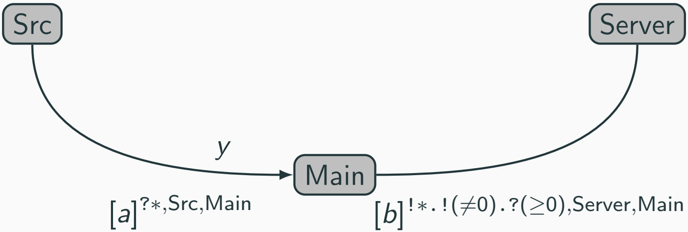
- C is the **contract** associated with u
- p identifies the guilty partner for values **received from** u
- q identifies the guilty partner for values **sent on** u

runtime monitoring: first-order example



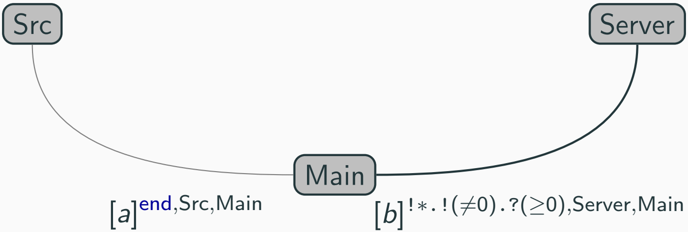

```
let main () = ...
  let x, a = recv a in
  let y, a = recv a in
  let b = send x b in
  let b = send y b in
  let w, b = recv b in ...
```

runtime monitoring: first-order example



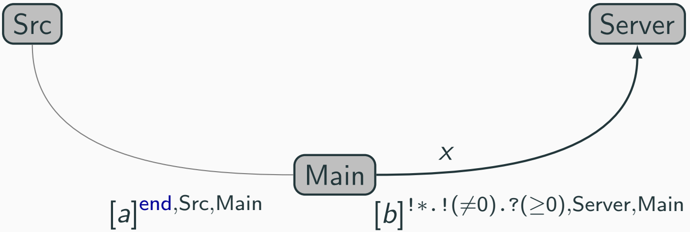

```
let main () = ...
  let x, a = recv a in
  let y, a = recv a in
  let b = send x b in
  let b = send y b in
  let w, b = recv b in ...
```

runtime monitoring: first-order example



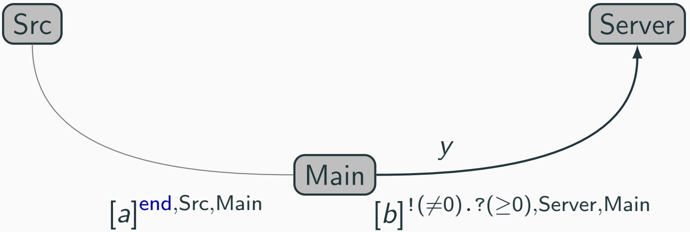

```
let main () = ...
  let x, a = recv a in
  let y, a = recv a in
  let b = send x b in
  let b = send y b in
  let w, b = recv b in ...
```

runtime monitoring: first-order example



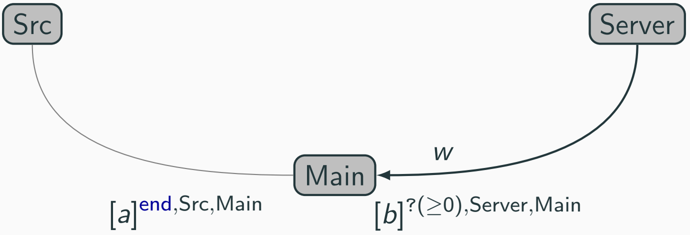

```
let main () = ...
  let x, a = recv a in
  let y, a = recv a in
  let b = send x b in
  let b = send y b in
  let w, b = recv b in ...
```

runtime monitoring: first-order example



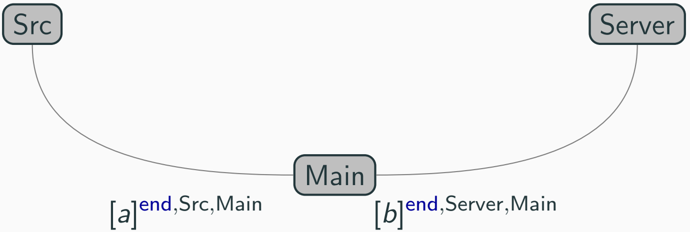

```
let main () = ...
  let x, a = recv a in
  let y, a = recv a in
  let b = send x b in
  let b = send y b in
  let w, b = recv b in ...
```

runtime monitoring: first-order example



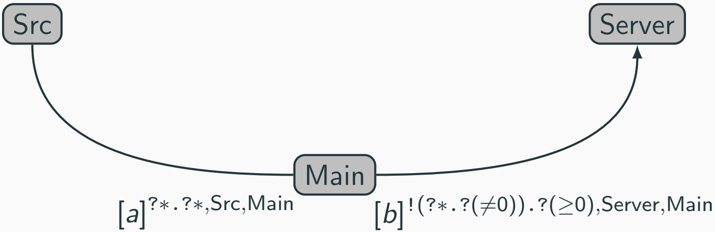

```
let main () = ...
  let x, a = recv a in
  let y, a = recv a in
  let b = send x b in
  let b = send y b in
  let w, b = recv b in ...
```

runtime monitoring: first-order example



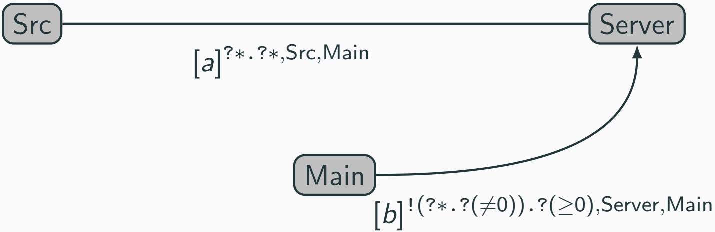

```
let main () = ...
  let x, a = recv a in
  let y, a = recv a in
  let b = send x b in
  let b = send y b in
  let w, b = recv b in ...
```

runtime monitoring: higher-order example



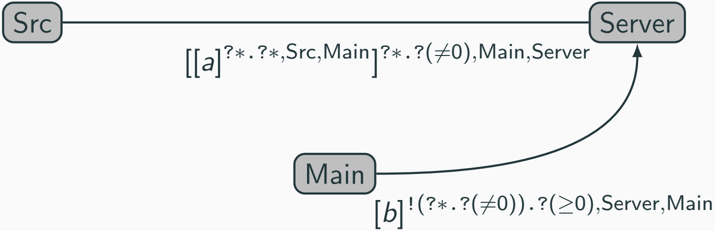

```
let main () = ...                  b : !(?int.?int).?int
  let b = send a b in              b : ?int
  let w, b = recv b in ...         b : end
```

runtime monitoring: higher-order example



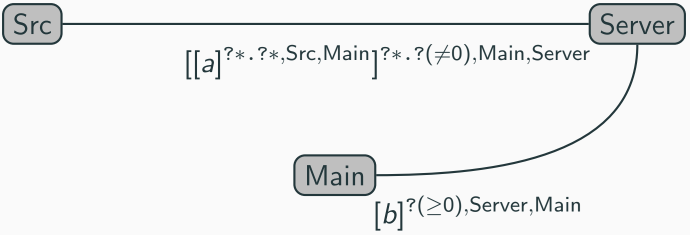

```
let main () = ...                  b : !(?int.?int).?int
  let b = send a b in              b : ?int
  let w, b = recv b in ...         b : end
```

runtime monitoring: higher-order example



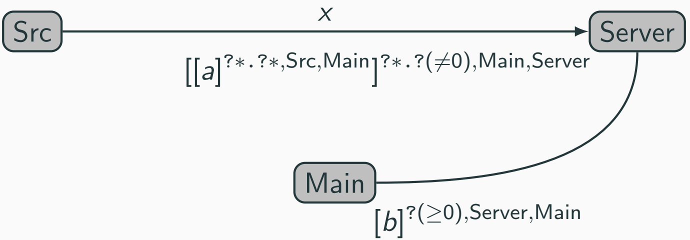

```
let main () = ...                 b : !(?int.?int).?int
  let b = send a b in             b : ?int
  let w, b = recv b in ...        b : end
```

runtime monitoring: higher-order example



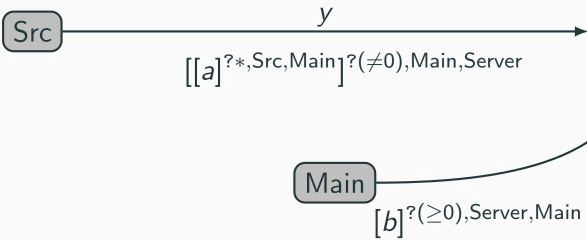

```
let main () = ...                   b : !(?int.?int).?int
  let b = send a b in               b : ?int
  let w, b = recv b in ...          b : end
```

runtime monitoring: higher-order example



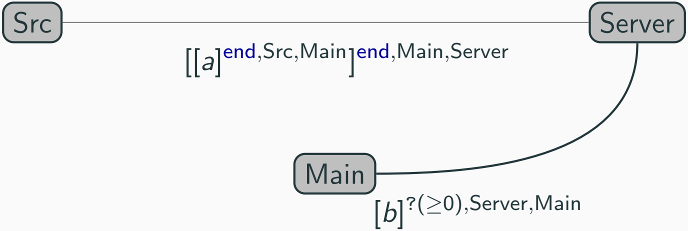

```
let main () = ...                  b : !(?int.?int).?int
  let b = send a b in              b : ?int
  let w, b = recv b in ...         b : end
```

runtime monitoring: higher-order example



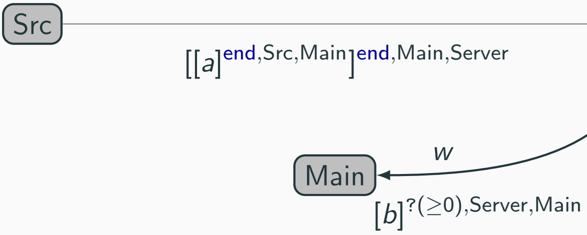

```
let main () = ...                b : !(?int.?int).?int
  let b = send a b in            b : ?int
  let w, b = recv b in ...       b : end
```

runtime monitoring: higher-order example




```
let main () = ...                b : !(?int.?int).?int
  let b = send a b in            b : ?int
  let w, b = recv b in ...       b : end
```

runtime monitoring: higher-order example




```
let main () = ...                b : !(?int.?int).?int
  let b = send a b in            b : ?int
  let w, b = recv b in ...       b : end
```

an example of dependent contract

```
let contract =
  send_d any_c @@ fun x →
  send_d (flat_c (≠ 0)) @@ fun y →
  recv_c (flat_c (fun w → x == (x / y) * y + w)) @@
  end_c
```

- contracts may **depend** on previously exchanged messages
- `send_c` is a degenerate version of `send_d`

blame correctness

Definition (local honesty)

A process is **locally honest** if it complies with the contracts **it is aware of** undecidable!

Theorem (blame correctness)

Locally honest processes are never blamed, even if they interact with dishonest processes

Concluding remarks

further developments

Safety properties

- protocol compliance (this talk)
- deadlock freedom

Liveness properties

- fair subtyping (aka fair testing, but for session types)
- lock freedom

Static linearity

- type inference for Linear Haskell (ongoing)

FuSe available from my home page

Thank You

References

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017.

Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *Proceedings of CONCUR'11*, LNCS 6901, pages 280–296. Springer, 2011.

Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, LNCS 2757, pages 439–453. Springer, 2002.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.

Hernán C. Melgratti and Luca Padovani. Chaperone contracts for higher-order sessions. *PACMPL*, 1(ICFP):35:1–35:29, 2017.

Luca Padovani. Context-free session type inference. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 804–830. Springer, 2017a.

Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017b.

Luca Padovani. Context-free session type inference. *ACM Trans. Program. Lang. Syst.*, 41(2):9:1–9:37, March 2019. ISSN 0164-0925.

Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of* *the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 462–475. ACM, 2016.