chaperone contracts for higher-order sessions

Hernán Melgratti
University of Buenos Aires
Argentina

Luca Padovani
University of Torino
Italy

sessions and session types

`!int.!int.?int` `?int.?int.!int`

- **session** = private communication channel with 2 endpoints
- **session type** = protocol specification as a type (≈ FSA)

pairing session types with contracts

Motivation

- finding more bugs and making it easier to locate their cause

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

1. send any number
2. send a number $n \neq 0$
3. receive a number $m > 0$

sessions as **non-uniform** mutable objects

Contracts for higher-order functions and mutable objects

- Findler & Felleisen, ICFP 2002
- Strickland, Tobin-Hochstadt, Findler & Flatt, OOPSLA 2012

higher-order functions $\iff$ higher-order sessions

`put` and `get` $\iff$ `send` and `recv`

Differences

- **order** of operations **constrained** by the session type
- **type** of messages **may change** over time
- **contract** of messages **may change** over time

a model of functions, sessions and contracts

functional core

$$x$$
$$\lambda x\,.\,e$$
$$e_1\ e_2$$

threads and sessions
[Gay & Vasconcelos, JFP 2010]

```
send
recv
...
```

monitors and blames
[Findler & Felleisen, ICFP 2002]

$$[e]^{C,p,q}$$
`blame` p

contracts for sessions
[this work]

```
send_c c d
recv_c c d
...
```

sample contract

```
!int.!int.?int
```

1. send any number
2. send a number $n \neq 0$
3. receive a number $m \geq 0$

```
send_c
  any_c
   (send_c
      (flat_c (≠ 0))
      (recv_c
         (flat_c (≥ 0))
        end_c))
```

sample contract

```
!int.!int.?int
```

1. send any number
2. send a number $n \neq 0$
3. receive a number $m \geq 0$

```
send_c
  any_c
   (send_c
      (flat_c (≠ 0))
      (recv_c
         (flat_c (≥ 0))
         end_c))
```

sample contract

```
!int.!int.?int
```

1. send any number
2. send a number $n \neq 0$
3. receive a number $m \geq 0$

```
send_c
  any_c
   (send_c
      (flat_c (≠ 0))
      (recv_c
         (flat_c (≥ 0))
         end_c))
```

sample program

```
let    a = connect server in        a : !int.!int.?int
let    a = send 1234 a in           a : !int.?int
let    a = send 56 a in             a : ?int
let m, a = recv a in                a : end
...
```

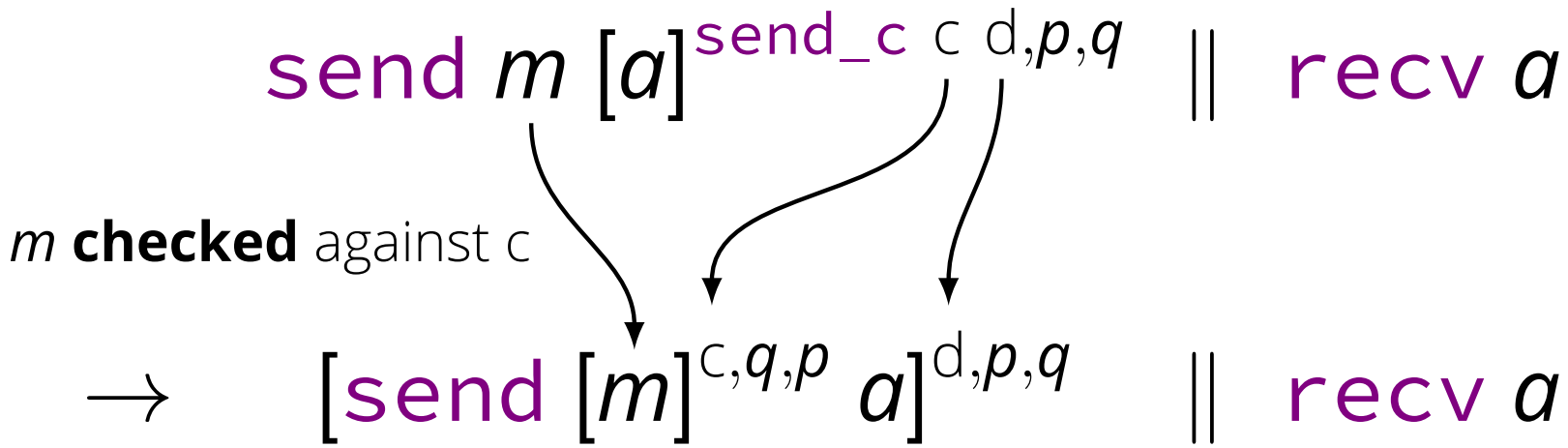
Communication primitives always return the endpoint being used

- the **type** of the endpoint is updated at each rebinding
- the **contract** of the endpoint can be updated as well!

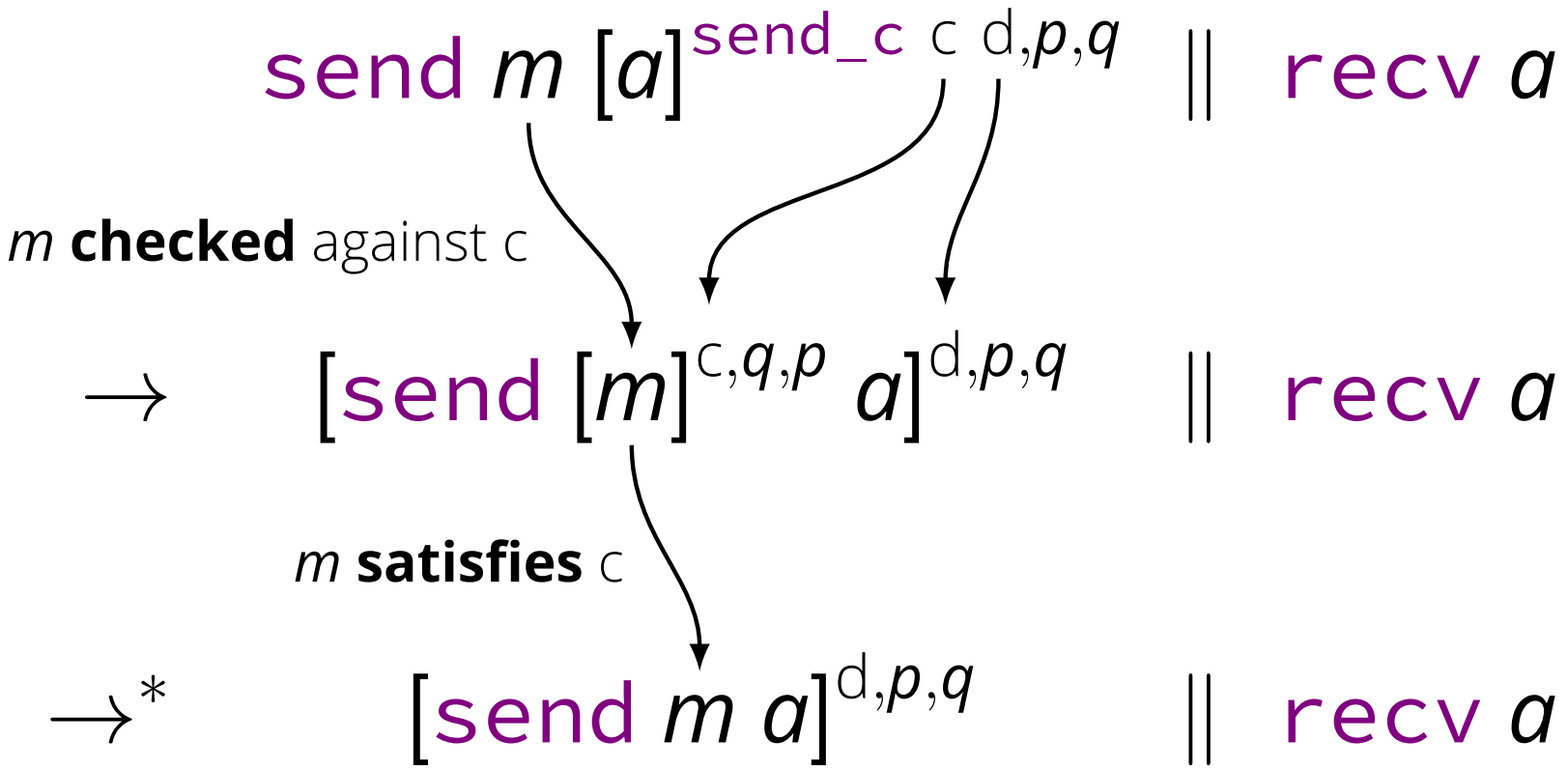
first-order monitored output

$$\texttt{send}\ m\ [a]^{\texttt{send\_c}\ \mathsf{c}\ \mathsf{d},p,q} \quad \| \quad \texttt{recv}\ a$$

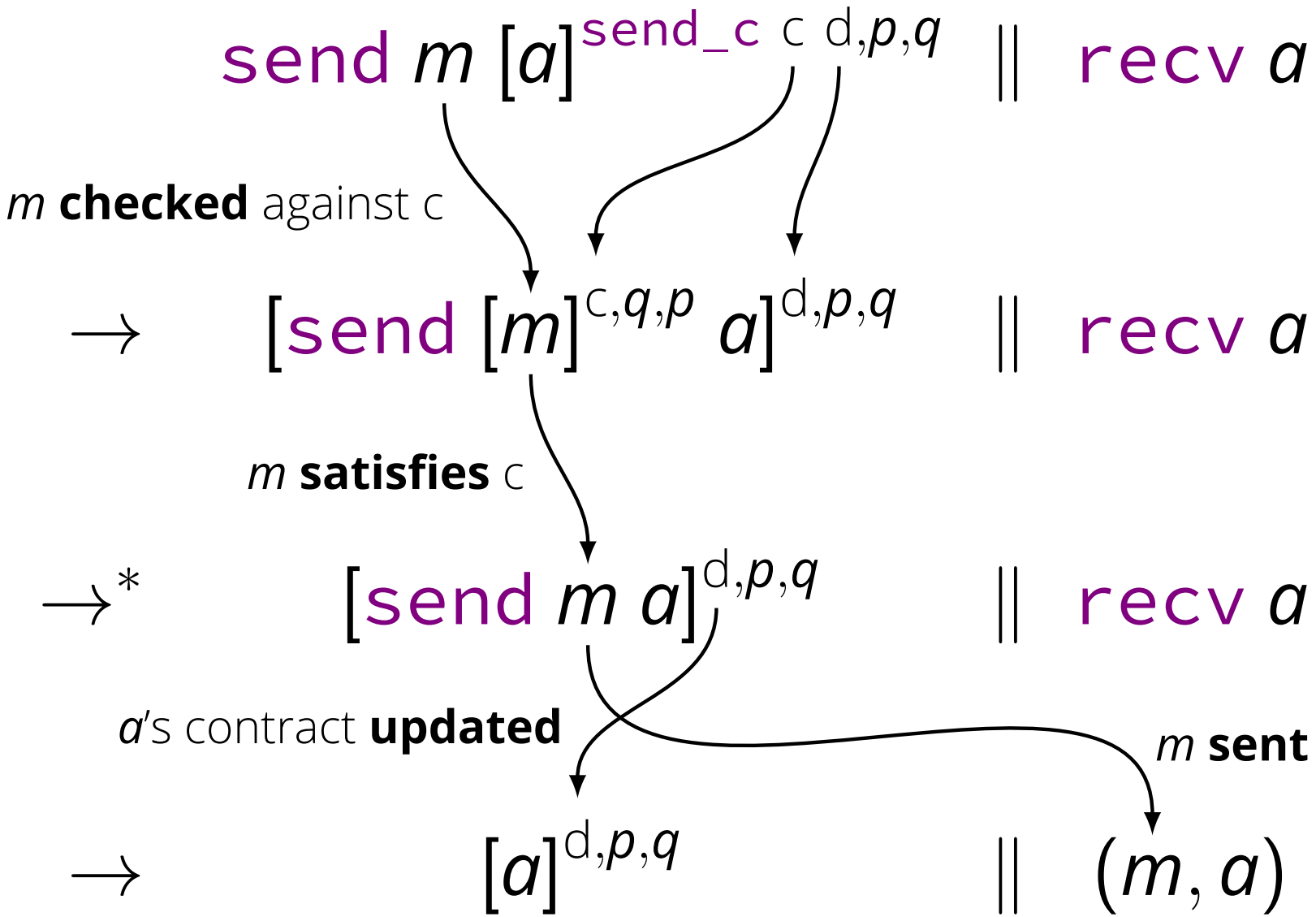
first-order monitored output



first-order monitored output



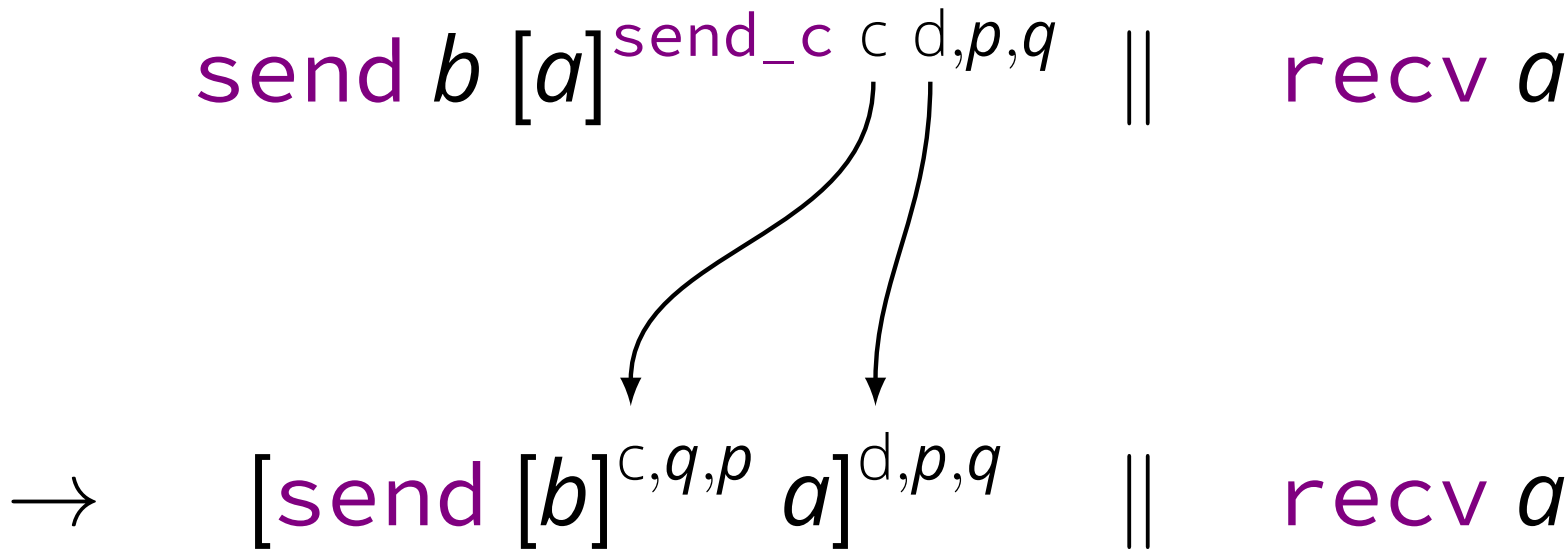
first-order monitored output



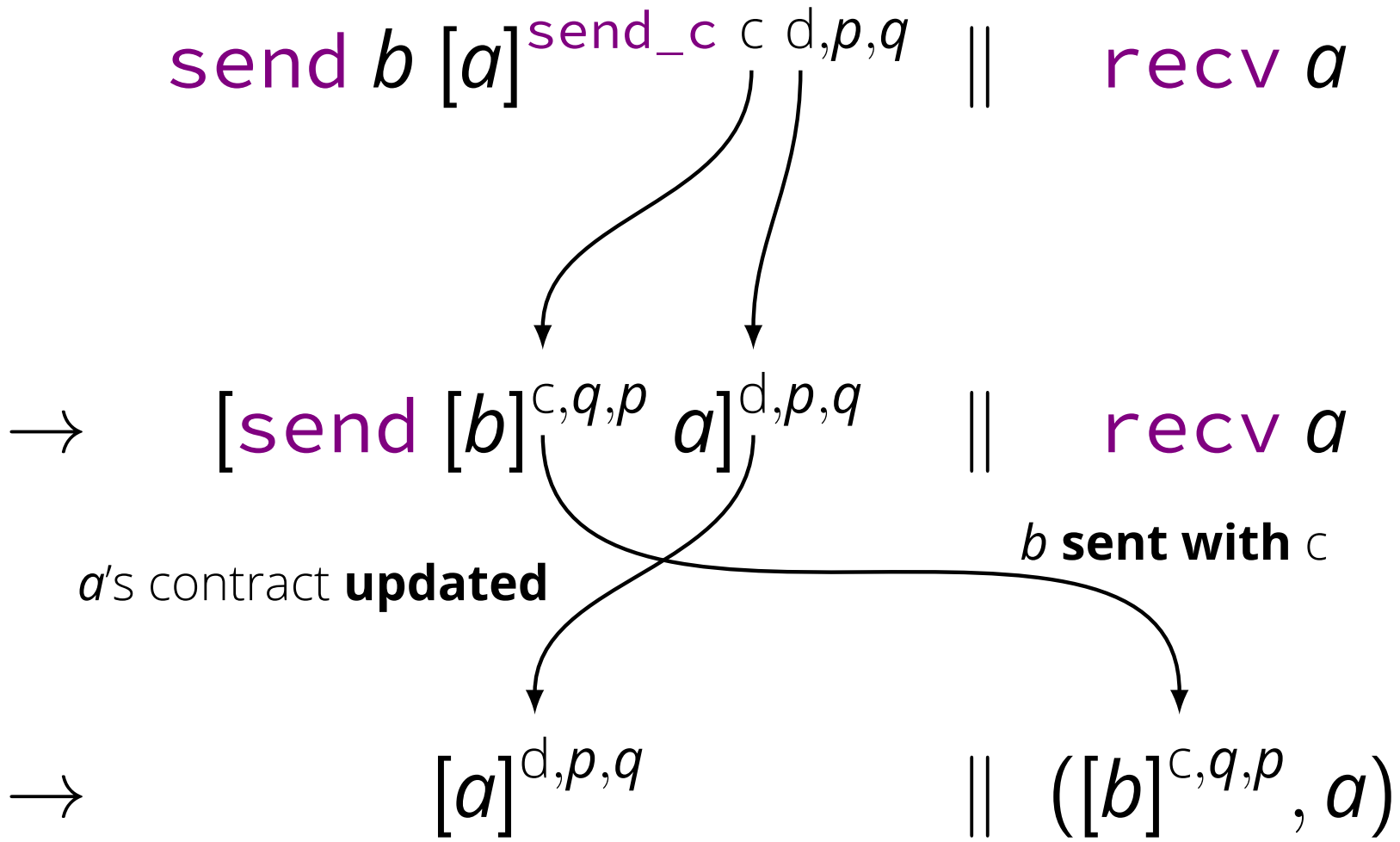
higher-order monitored output

$$\texttt{send}\ b\ [a]^{\texttt{send\_c}\ \mathsf{c}\ \mathsf{d},p,q} \quad \| \quad \texttt{recv}\ a$$

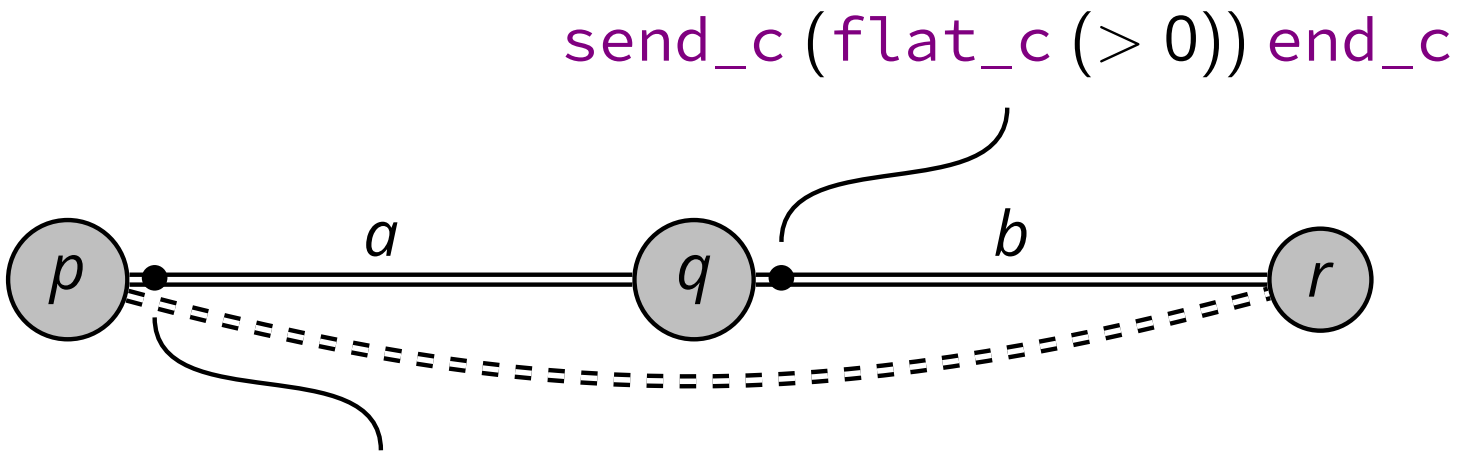
higher-order monitored output



higher-order monitored output



who gets the blame?



- q sends b to p
- p **thinks** that it is safe to send **any** number, *e.g.* -1, on b
- r **expects** to receive a positive number from p
- by the time the violation is detected, q is no longer involved

honest modules won't be blamed

Local compliance ⇒ Global blame freedom

If p complies with the contracts it **knows**, then no one will blame p

It's difficult to formalize what "knowing a contract" means

- contracts are decomposed and turned inside out
- not all endpoints have a contract
- ...

We prove the result using an alternative semantics

- makes it easy to identify syntactically the contracts known by p
- we show that the two semantics are essentially equivalent

honest modules won't be blamed

Local compliance ⇒ Global blame freedom

If p complies with the contracts it **knows**, then no one will blame p

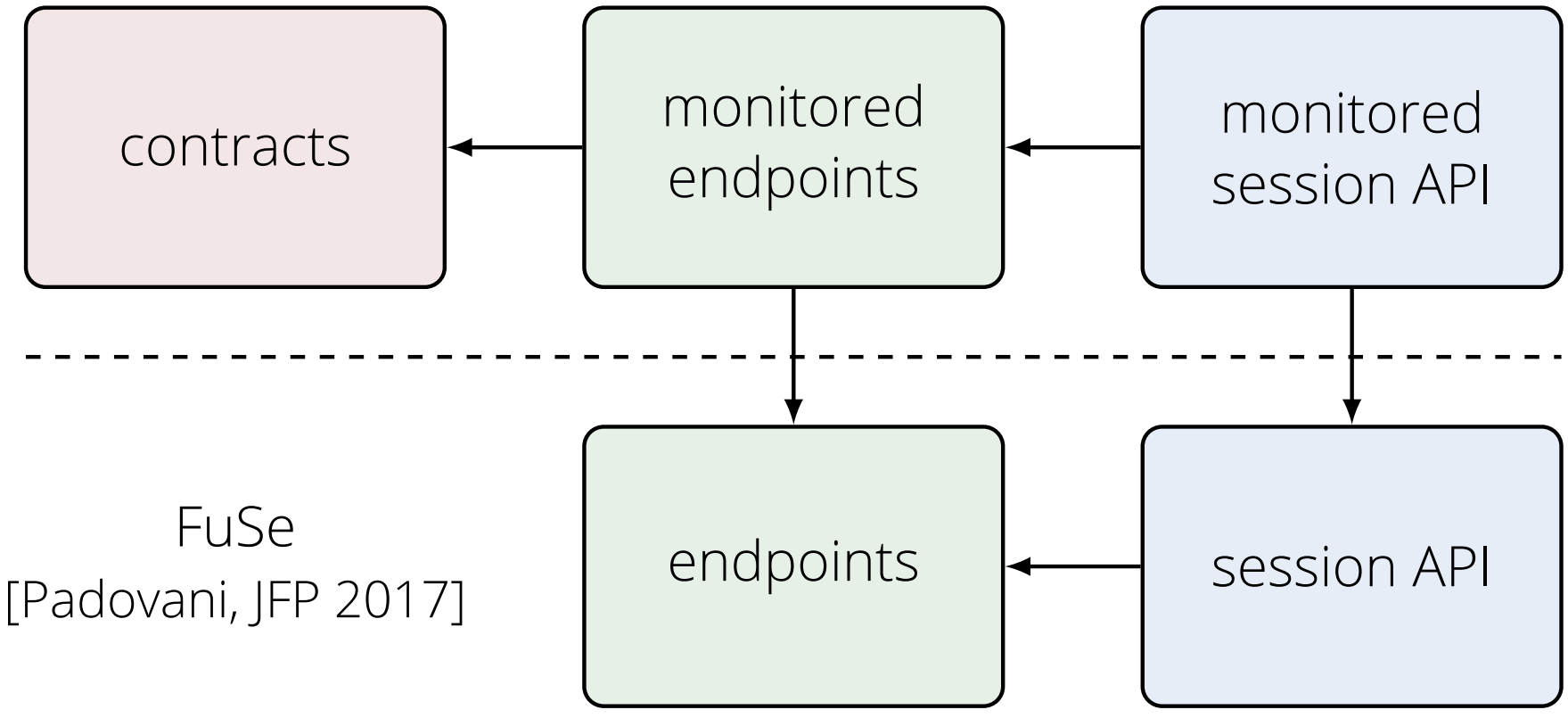
It's difficult to formalize what "knowing a contract" means

- contracts are decomposed and turned inside out
- not all endpoints have a contract
- …

We prove the result using an alternative semantics

- makes it easy to identify syntactically the contracts known by p
- we show that the two semantics are essentially equivalent

an OCaml implementation



FuSe
[Padovani, JFP 2017]

- modular design, portable to other session libraries

summary

Contributions

- monitoring system for sessions with dynamic contract update
- blame correctness

Other features

- **dependent** contracts
- contracts for **recursive** and **branching** protocols

Open issues

- functions in messages cannot carry session endpoints
- …

THANKS!