Session-ocaml:

a Session-based Library with

Polarities and **Lenses**

Keigo Imai
Gifu University, JP

Nobuko Yoshida
Imperial College London, UK

Shoji Yuen
Nagoya University, JP

COORDINATION 2017

Neuchâtel, Switzerland

20th June, 2017

Introduction

- Implementation of distributed software is notoriously difficult
- OCaml: a concise language with fast runtime
- Various concurrent/distributed applications
  - High freq. trading in Jane Street Capital
  - Ocsigen/Eliom [web server/framework], BuckleScript [translates to JavaScript]
  - MirageOS, MLDonkey [P2P]
- Aim: to give a **static assuarance** for communicating software

Session-ocaml: A Session Type implementation in OCaml

- Session types guarantee communication safety and session fidelity **in OCaml**

- Two novel features:

#1. Session-Type (Duality) Inference

→ Equality-based duality checking by **polarised session types**

#2. Linearity in (non-linear) OCaml types

→ Statically-typed ***delegation*** with **slot-oriented programming**

Session-ocaml in a Nutshell: (1) Session-type inference

Session-ocaml program:

```
let main () =
 send "Hello" >>
 let%s x = recv () in
 close ()
```

#1:

Session-type inference solely done by OCaml compiler

via

"Polarised session types"

(explained later)

Type signature (inferred):

```
val main:
  req[string];resp[τ];close
```

(much simplified than reality)

Session-ocaml in a Nutshell: (2) Linearity by slot-oriented programming

GV-style session programming:

(in FuSe [Padovani'16] and GVinHS [Lindley&Morris,'16])

```
let s'      = send s "Hello" in
let x, s'' = recv s' in
close s''
```

1. A new session endpoint is created for each communication step
2. Every endpoint must be <u>linearly used</u> (not checkable by OCaml types)

Slot-oriented session programming:

(in Session-ocaml)

```
send _0 "Hello" >>
let%s x = recv _0 in
close _0
```

Session-ocaml in a Nutshell: (2) Linearity by slot-oriented programming

GV-style session programming:

(in FuSe [Padovani'16] and GVinHS [Lindley&Morris,'16])

```
let s'     = send s "Hello" in
let x, s'' = recv s' in
close s''
```

1. A new session endpoint is created for each communication step
2. Every endpoint must be <u>linearly used</u> (not checkable by OCaml types)

Slot-oriented session programming:

(in Session-ocaml)

a few syntactic extensions

use monads

```
send _0 "Hello" >>
let%s x = recv _0 in
close _0
```

Session-ocaml in a Nutshell: (2) Linearity by slot-oriented programming

GV-style session programming:

(in FuSe [Padovani'16] and GVinHS [Lindley&Morris,'16])

```
let s'      = send s "Hello" in
let x, s'' = recv s' in
close s''
```

Slot-oriented session programming:

(in Session-ocaml)

```
send _0 "Hello" >>
let%s x = recv _0 in
close _0
```

a few syntactic extensions

use monads

Use "slot" numbers (_0, _1,..) as place-holders

1. A new session endpoint is created for each communication step
2. Every endpoint must be linearly used (not checkable by OCaml types)

#2: Provides linearity on top of NON-linear type system

History of Session type implementation (in Haskell & OCaml)

Haskell

Neubauer & Thiemann '06

Pucella & Tov, '08

Sackman & Eisenbach, '08

Imai, Yuen & Agusa, '10

Orchard & **Yoshida**, '16

Lindley & Morris, '16

OCaml

Pucella & Tov, '08

Padovani, '16

Imai, Yoshida & Yuen, '17

Very few OCaml-based session types --

Duality and Linearity were the major obstacles

(which Haskell coped with various type-level features)

History of Session type implementation (in Haskell & OCaml)

Haskell

Neubauer & Thiemann '06

Pucella & Tov, '08

Sackman & Eisenbach, '08

Imai, Yuen & Agusa, '10

Orchard & **Yoshida**, '16

Lindley & Morris, '16

OCaml

Pucella & Tov, '08

FuSe

Padovani, '16

Imai, Yoshida & Yuen, '17

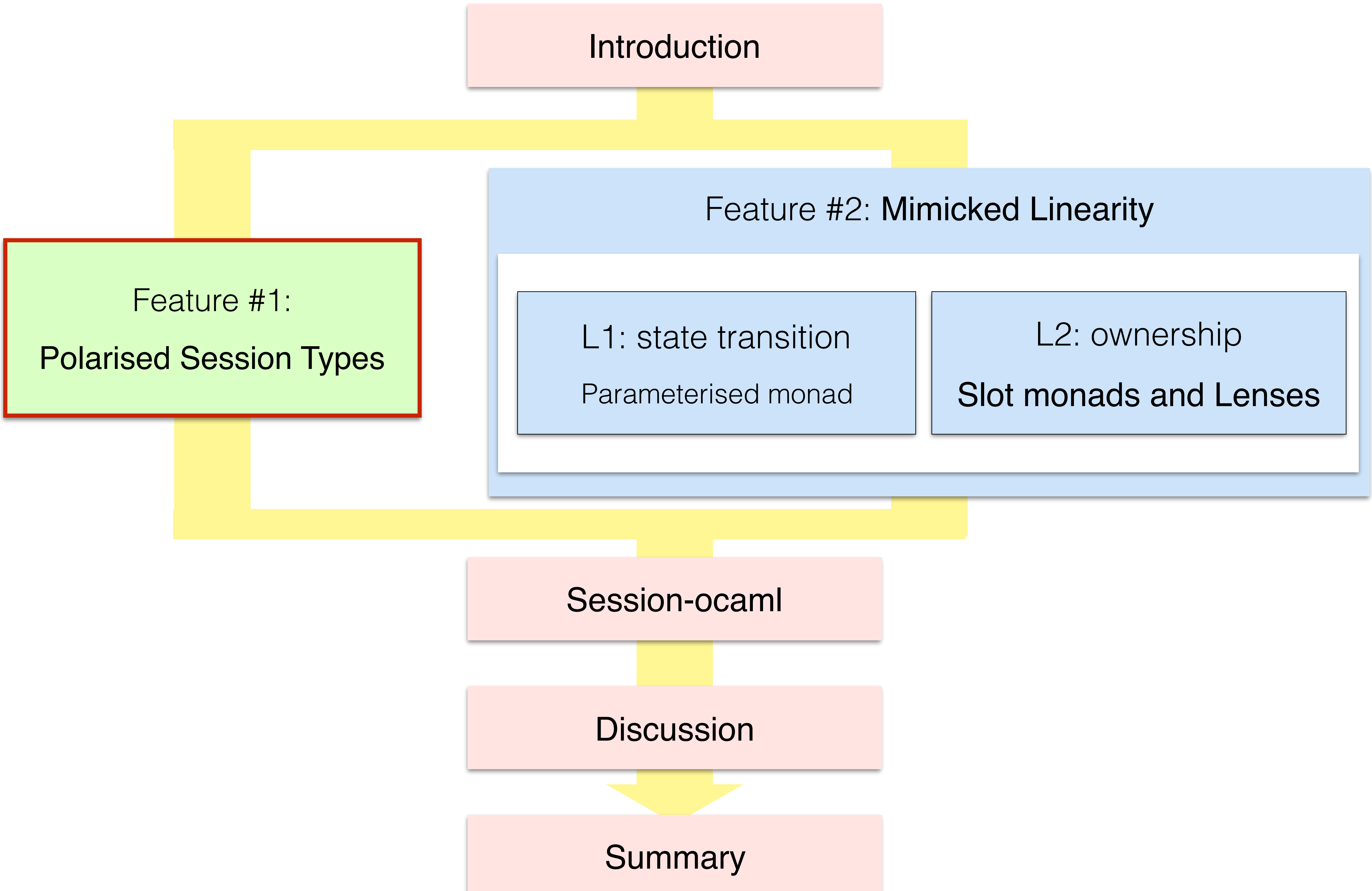
Session-ocaml

Very few OCaml-based session types --

Duality and Linearity were the major obstacles

(which Haskell coped with various type-level features)

Presentation structure



Original session types and polarised session types

Original session types [Honda '97]:

```
!int;close
?string;!bool;close
μα.!ping;?pong;α
```

Duality:

$$\overline{!v;S} = ?v;\overline{S} \qquad \overline{\&\{l_i : S_i\}} = \oplus\{l_i : \overline{S_i}\}$$
$$\overline{?v;S} = !v;\overline{S} \qquad \overline{\oplus\{l_i : S_i\}} = \&\{l_i : \overline{S_i}\}$$
$$\overline{\mu\alpha.S} = \mu\alpha.\overline{S}[\overline{\alpha}/\alpha] \qquad \overline{\texttt{close}} = \texttt{close}$$

Duality is <u>too complex</u> to have in OCaml type

Polarised session types:

$\texttt{req[int];close}^{\texttt{cli}}$

$\texttt{resp[string];req[bool];close}^{\texttt{cli}}$

$\texttt{μα.resp[ping];req[pong];α}^{\texttt{serv}}$

Duality:

$$\overline{P^{\texttt{serv}}} = P^{\texttt{cli}} \qquad \overline{P^{\texttt{cli}}} = P^{\texttt{serv}}$$

Original session types and polarised session types

Original session types [Honda '97]:

```
!int;close
?string;!bool;close
μα.!ping;?pong;α
```

Polarised session types:

$\texttt{req[int];close}^{\texttt{cli}}$

$\texttt{resp[string];req[bool];close}^{\texttt{cli}}$

$\mu\alpha.\texttt{resp[ping];req[pong];}\alpha^{\texttt{serv}}$

Use $\{$**req**, **resp**$\}$ **instead of** $\{!, ?\}$

Duality:

$$\overline{!v;S} = ?v;\overline{S} \qquad \overline{\&\{l_i : S_i\}} = \oplus\{l_i : \overline{S_i}\}$$
$$\overline{?v;S} = !v;\overline{S} \qquad \overline{\oplus\{l_i : S_i\}} = \&\{l_i : \overline{S_i}\}$$
$$\overline{\mu\alpha.S} = \mu\alpha.\overline{S[\overline{\alpha}/\alpha]} \qquad \overline{\texttt{close}} = \texttt{close}$$

Duality:

$$\overline{P^{\texttt{serv}}} = P^{\texttt{cli}} \qquad \overline{P^{\texttt{cli}}} = P^{\texttt{serv}}$$

Duality is too complex to have in OCaml type

Original session types and polarised session types

Original session types [Honda '97]:

```
!int;close
?string;!bool;close
μα.!ping;?pong;α
```

Polarised session types:

Polarity {cli, serv} **gives** **modality**

$\texttt{req[int];close}^{\texttt{cli}}$

$\texttt{resp[string];req[bool];close}^{\texttt{cli}}$

$\texttt{μα.resp[ping];req[pong];α}^{\texttt{serv}}$

Use {**req**, **resp**} **instead of** {!,?}

Duality:

$$\overline{!v;S} = ?v;\overline{S} \qquad \overline{\&\{l_i : S_i\}} = \oplus\{l_i : \overline{S_i}\}$$
$$\overline{?v;S} = !v;\overline{S} \qquad \overline{\oplus\{l_i : S_i\}} = \&\{l_i : \overline{S_i}\}$$
$$\overline{\mu\alpha.S} = \mu\alpha.\overline{S[\overline{\alpha}/\alpha]} \qquad \overline{\texttt{close}} = \texttt{close}$$

Duality:

$$\overline{P^{\texttt{serv}}} = P^{\texttt{cli}} \qquad \overline{P^{\texttt{cli}}} = P^{\texttt{serv}}$$

Duality is too complex to have in OCaml type

Original session types and polarised session types

Original session types [Honda '97]:

```
!int;close
?string;!bool;close
μα.!ping;?pong;α
```

Use {req, resp} instead of {!,?}

Polarised session types:

Polarity {cli, serv} gives modality

$\texttt{req[int];close}^{\texttt{cli}}$

$\texttt{resp[string];req[bool];close}^{\texttt{cli}}$

$\mu\alpha.\texttt{resp[ping];req[pong];}\alpha^{\texttt{serv}}$

Duality:

$$\overline{!v;S} = ?v;\overline{S} \qquad \overline{\&\{l_i : S_i\}} = \oplus\{l_i : \overline{S_i}\}$$
$$\overline{?v;S} = !v;\overline{S} \qquad \overline{\oplus\{l_i : S_i\}} = \&\{l_i : \overline{S_i}\}$$
$$\overline{\mu\alpha.S} = \mu\alpha.\overline{S[\overline{\alpha}/\alpha]} \qquad \overline{\texttt{close}} = \texttt{close}$$

Duality:

$$\overline{P^{\texttt{serv}}} = P^{\texttt{cli}} \qquad \overline{P^{\texttt{cli}}} = P^{\texttt{serv}}$$

Duality is <u>too complex</u> to have in OCaml type

Duality is much **simpler** and **type-inference friendly**

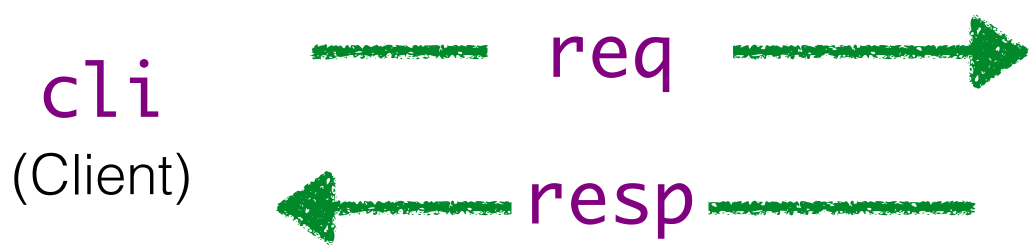
Session-type inference in Session-ocaml

```
let eqclient () =
  connect_ eqch (fun () ->
    send (123, 456) >>
    let%s ans = recv () in
    close ()) ()
```

Session-type inference in Session-ocaml

proactive

```
let eqclient () =
  connect_ eqch (fun () ->
    send (123, 456) >>
    let%s ans = recv () in
    close ()) ()
```



Session-type inference in Session-ocaml

```
let eqclient () =
  connect_ eqch (fun () ->
    send (123, 456) >>
    let%s ans = recv () in
    close ()) ()
```

proactive

```
(req[int*int];
 resp[bool];
 close)^cli
```

cli
(Client)

req →

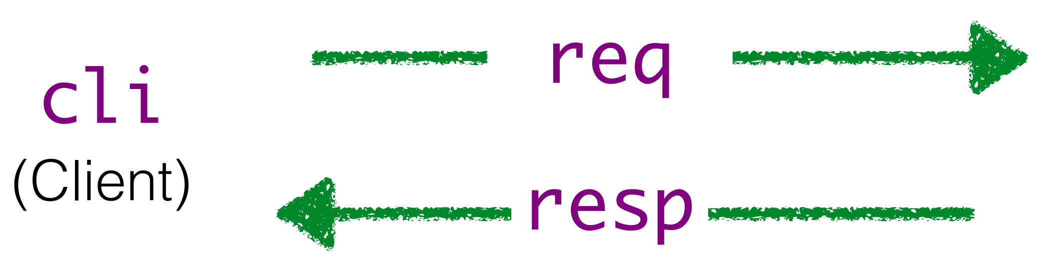
← resp

Session-type inference in Session-ocaml

```
let eqclient () =
  connect_ eqch (fun () ->
    send (123, 456) >>
    let%s ans = recv () in
    close ()) ()
```

proactive

```
(req[int*int];
 resp[bool];
 close)cli
```



inferred

```
val eqch: req[int*int];resp[bool];close
```

(protocol type)

Session-type inference in Session-ocaml

```
let eqclient () =
  connect_ eqch (fun () ->
    send (123, 456) >>
    let%s ans = recv () in
    close ()) ()
```

```
let eqserv () =
  accept_ eqch (fun () ->
    let%s x,y = recv () in
    send (x=y) >>
    close ())) ()
```

proactive

```
(req[int*int];
 resp[bool];
 close)cli
```

cli
(Client)

req

resp

inferred

```
val eqch: req[int*int];resp[bool];close
```

(protocol type)

Session-type inference in Session-ocaml

```
let eqclient () =
  connect_ eqch (fun () ->
    send (123, 456) >>
    let%s ans = recv () in
    close ()) ()
```

proactive

```
let eqserv () =
  accept_ eqch (fun () ->
    let%s x,y = recv () in
    send (x=y) >>
    close ())) ()
```

reactive

(req[int*int];
resp[bool];
close)^{cli}

cli
(Client)

req →

← resp

serv
(Server)

inferred

```
val eqch: req[int*int];resp[bool];close
```

(protocol type)

Session-type inference in Session-ocaml

```
let eqclient () =
  connect_ eqch (fun () ->
    send (123, 456) >>
    let%s ans = recv () in
    close ()) ()
```

proactive

```
let eqserv () =
  accept_ eqch (fun () ->
    let%s x,y = recv () in
    send (x=y) >>
    close ())) ()
```

reactive

(req[int*int];
resp[bool];
close)$^{\mathtt{cli}}$

cli
(Client)

req

resp

serv
(Server)

(req[int*int];
resp[bool];
close)$^{\mathtt{serv}}$

inferred

```
val eqch: req[int*int];resp[bool];close
```

(protocol type)

Session-type inference in Session-ocaml

```
let eqclient () =
  connect_ eqch (fun () ->
    send (123, 456) >>
    let%s ans = recv () in
    close ()) ()
```

proactive

```
let eqserv () =
  accept_ eqch (fun () ->
    let%s x,y = recv () in
    send (x=y) >>
    close ())) ()
```

reactive

$(\texttt{req[int*int]; resp[bool]; close})^{\texttt{cli}}$

cli
(Client)

req →

← resp

serv
(Server)

$(\texttt{req[int*int]; resp[bool]; close})^{\texttt{serv}}$

inferred

duality is checked by type equality

```
val eqch: req[int*int];resp[bool];close
```

(protocol type)

Branching & recursion

```
let rec eq_loop () =
  match%branch () with
  | `bin -> let%s x,y = recv () in
            send (x=y) >>=
            eq_loop
  | `fin -> close ()
in accept_ eqch2 eq_loop ()
```

Branching & recursion

```
let rec eq_loop () =
  match%branch () with
  | `bin -> let%s x,y = recv () in
            send (x=y) >>=
            eq_loop
  | `fin -> close ()
in accept_ eqch2 eq_loop ()
```

```
val eqch2: μα.req{ bin: req[int*int];resp[bool];α,
                   fin: close }
```

Branching & recursion

```
let rec eq_loop () =
  match%branch () with
  | `bin -> let%s x,y = recv () in
            send (x=y) >>=
            eq_loop
  | `fin -> close ()
in accept_ eqch2 eq_loop ()
```

```
val eqch2: μα.req{ bin: req[int*int];resp[bool];α,
                    fin: close }
```

Branching & recursion

```
let rec eq_loop () =
  match%branch () with
  | `bin -> let%s x,y = recv () in
            send (x=y) >>=
            eq_loop
  | `fin -> close ()
in accept_ eqch2 eq_loop ()
```

```
val eqch2: μα.req{ bin: req[int*int];resp[bool];α,
                   fin: close }
```

Session type subtyping in Session-ocaml

- OCaml's **polymorphic variant types** [Garrigue '00] simulates **subtyping** on session-type branchings

```
match%branch () with
| `bin -> ...
| `fin -> close ()
```

(selects `fin` label)

```
[%select () `fin]
```

Session type subtyping in Session-ocaml

- OCaml's **polymorphic variant types** [Garrigue '00] simulates **subtyping** on session-type branchings

```
match%branch () with
| `bin -> ...
| `fin -> close ()
```

inferred

(Session-ocaml type in the actual OCaml syntax)

```
[`branch of req * [`bin of ...
                  |`fin of [`close]]]
```

(selects `fin` label)

```
[`branch of req * [> `fin of [`close]]]
```

inferred

```
[%select () `fin]
```

Session type subtyping in Session-ocaml

- OCaml's **polymorphic variant types** [Garrigue '00] simulates **subtyping** on session-type branchings

```
match%branch () with
| `bin -> ...
| `fin -> close ()
```

inferred

(Session-ocaml type in the actual OCaml syntax)

```
[`branch of req * [`bin of ...
                  |`fin of [`close]]]
```

(selects `fin` label)

```
[`branch of req * [> `fin of [`close]]]
```

inferred

"open" variant type

```
[%select () `fin]
```

Session type subtyping in Session-ocaml

- OCaml's **polymorphic variant types** [Garrigue '00] simulates **subtyping** on session-type branchings

```
match%branch () with
| `bin -> ...
| `fin -> close ()
```

inferred

(Session-ocaml type in the actual OCaml syntax)

```
[`branch of req * [`bin of ...
                  |`fin of [`close]]]
```

(selects `fin` label)

OCaml's type unification checks subsumption!

```
[`branch of req * [> `fin of [`close]]]
```

"open" variant type

inferred

```
[%select () `fin]
```

Small caveat in polarised session types

- **Problem:** two types for one modality

`send 100` has either type:

$\texttt{req[int];close}^{\texttt{cli}}$

or

$\texttt{resp[int];close}^{\texttt{serv}}$

depending on the polarity.

Small caveat in polarised session types

- **Problem:** two types for one modality

`send 100` has either type: `req[int];close`$^{\texttt{cli}}$ or `resp[int];close`$^{\texttt{serv}}$

depending on the polarity.

- **(Partial) Solution:** Polarity polymorphism!

$$\texttt{send 100} : \forall \gamma_1 \gamma_2 . \gamma_1 \texttt{[int];close}^{\gamma_1 * \gamma_2}$$

where

```
cli  ≡ req*resp
serv ≡ resp*req
```

("partial" since OCaml only allow ∀ at the prenex-position, though we think it works fine in many cases)

Comparing with FuSe's duality [Padovani, '16]

- Duality in FuSe [Padovani, '16]:

$$\overline{(\alpha, \beta)\ \mathtt{t}} = (\overline{\beta}, \alpha)\ \mathtt{t} \quad \text{(Dardha's encoding ['12])}$$

- Quite simple, however, nesting `t`'s becomes quite cumbersome to read by humans:

```
(binop * ((bool*bool) * (_0, bool*(_0,_0) t) t,_0) t, _0) t
```

(hence FuSe comes with **"type decoder"** Rosetta.)

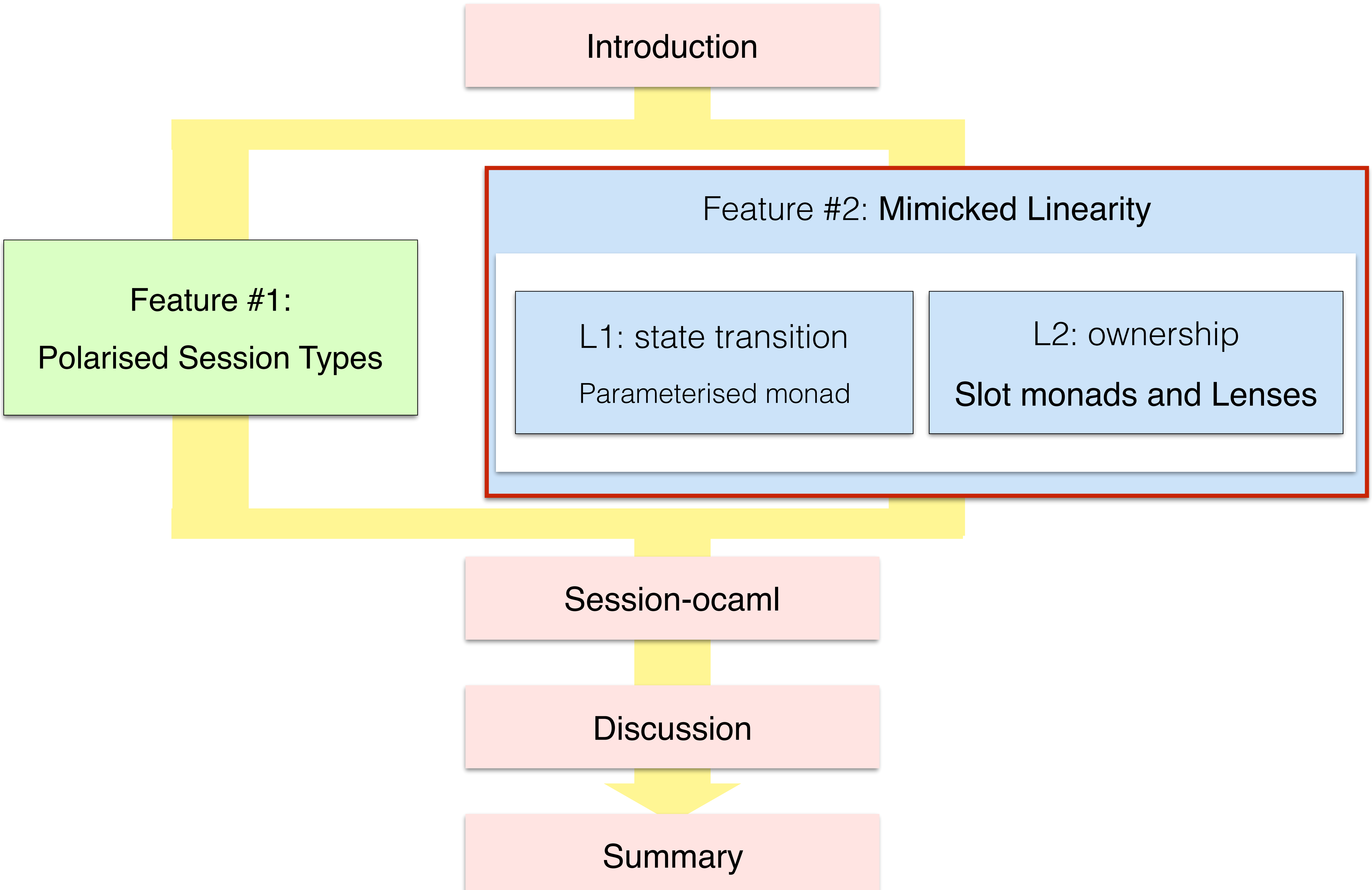
- Equivalent protocol type in Session-ocaml would be:

```
[`msg of req * binop * [`msg of req * (bool*bool) * [`msg of resp * bool * [`close]]]]
```

(Session-ocaml type in the actual OCaml syntax)

which is a bit longer, but much more understandable due to its "prefixing" manner.

Presentation structure

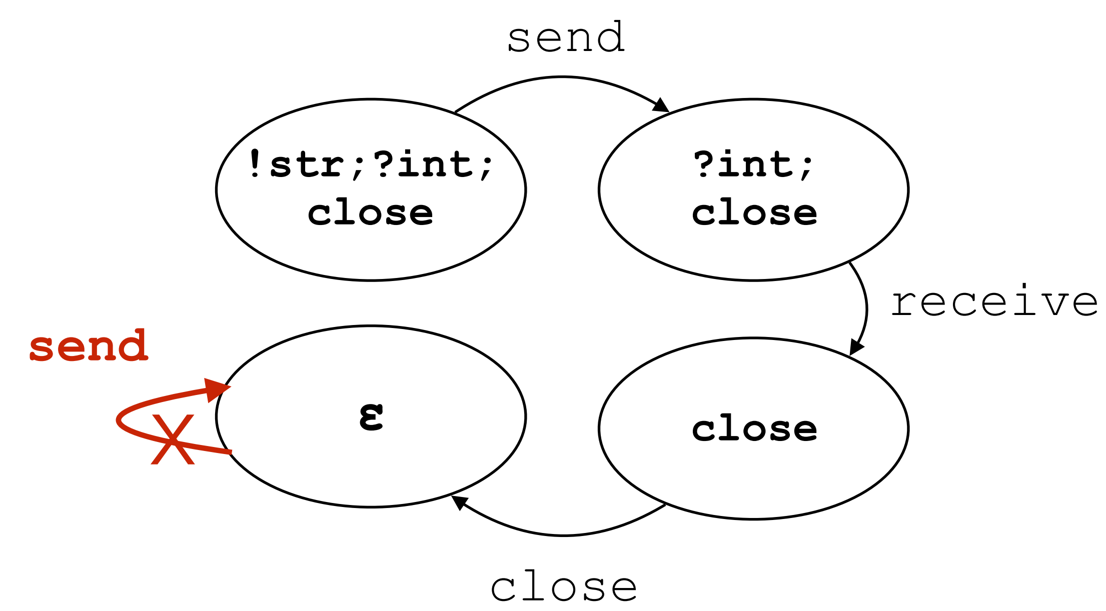


Linearity in session types is two-fold

(L1) **Enforcing state transition** in types

```
let s1 = send "Hello" s0 in
let s2, x = recv s1 in
close s2;
send s2 "Blah"
```

s2 is closed



(L2) **Tracking ownership** of a session endpoint

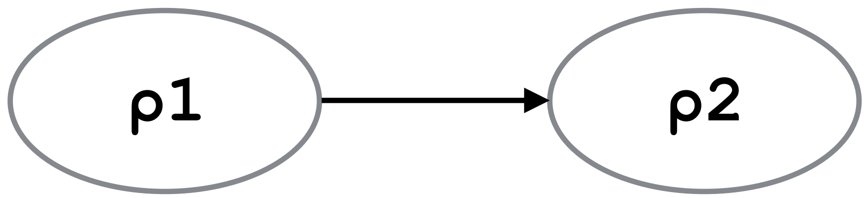
```
let s1 = delegate s0 t0 in
send t0 "Blah"
```

ownership of t0 is **transferred** to other thread

Solution to (L1): use a parameterised monad [Neubauer and Thiemann, '06]

```
type (ρ1, ρ2, τ) monad
```

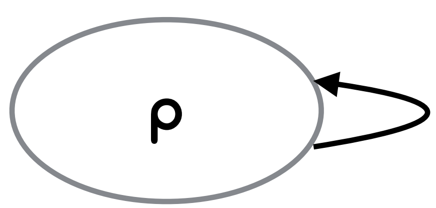
is a type of an effectful computation with state transition:



with return value of type τ.

```
val return : α -> (ρ, ρ, α) monad
```

is a "pure" (i.e. effect-less) computation with no state transition:



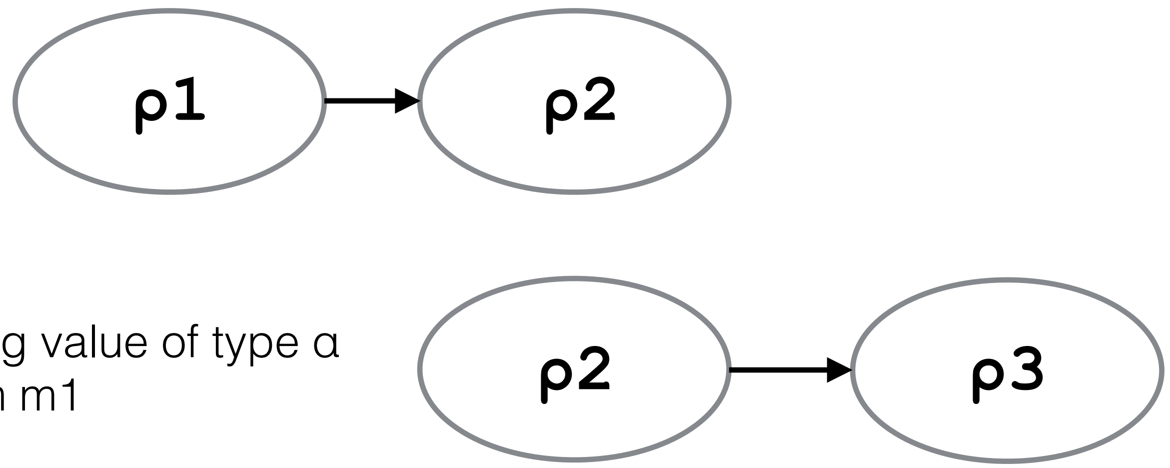
A parameterised monad (cont.)

```
val (>>=) : (ρ1, ρ2, α) monad -> (α -> (ρ2, ρ3, β) monad)
         -> (ρ1, ρ3, β) monad
```

combines two actions:

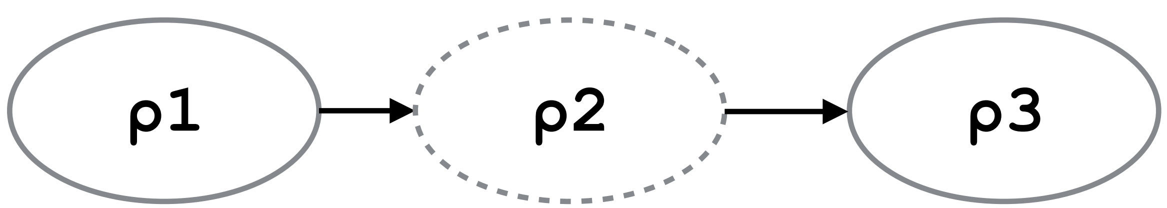
m1: with return value of type α

m2: using value of type α from m1 — with return value of type β



into:

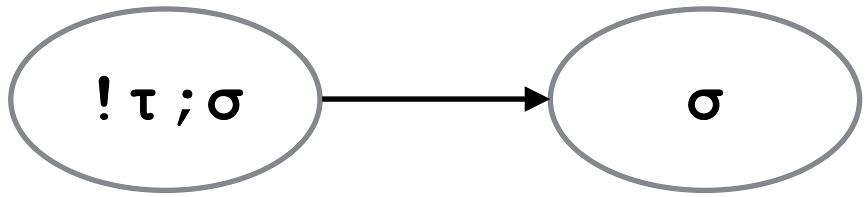
m1 >>= m2: with return value of type β



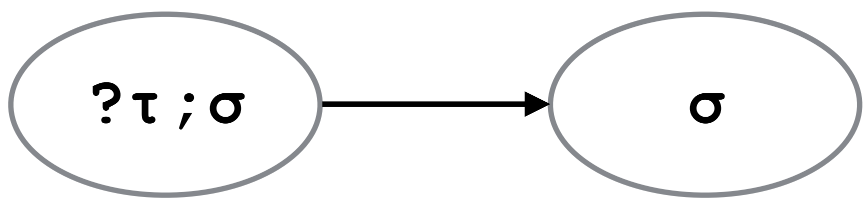
Session types as state transitions

The parameterised monad serves part of **Linearity (L1) in session types**:

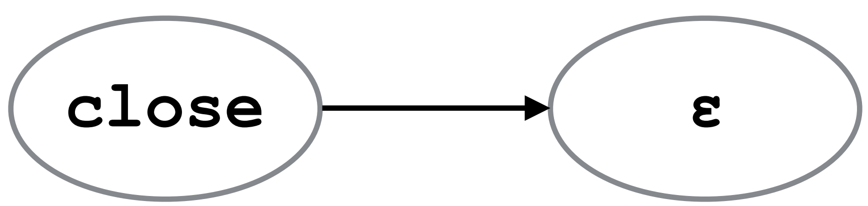
```
val send : τ -> (!τ;α, α, unit) monad
```




```
val recv : (?τ;α, α, τ) monad
```

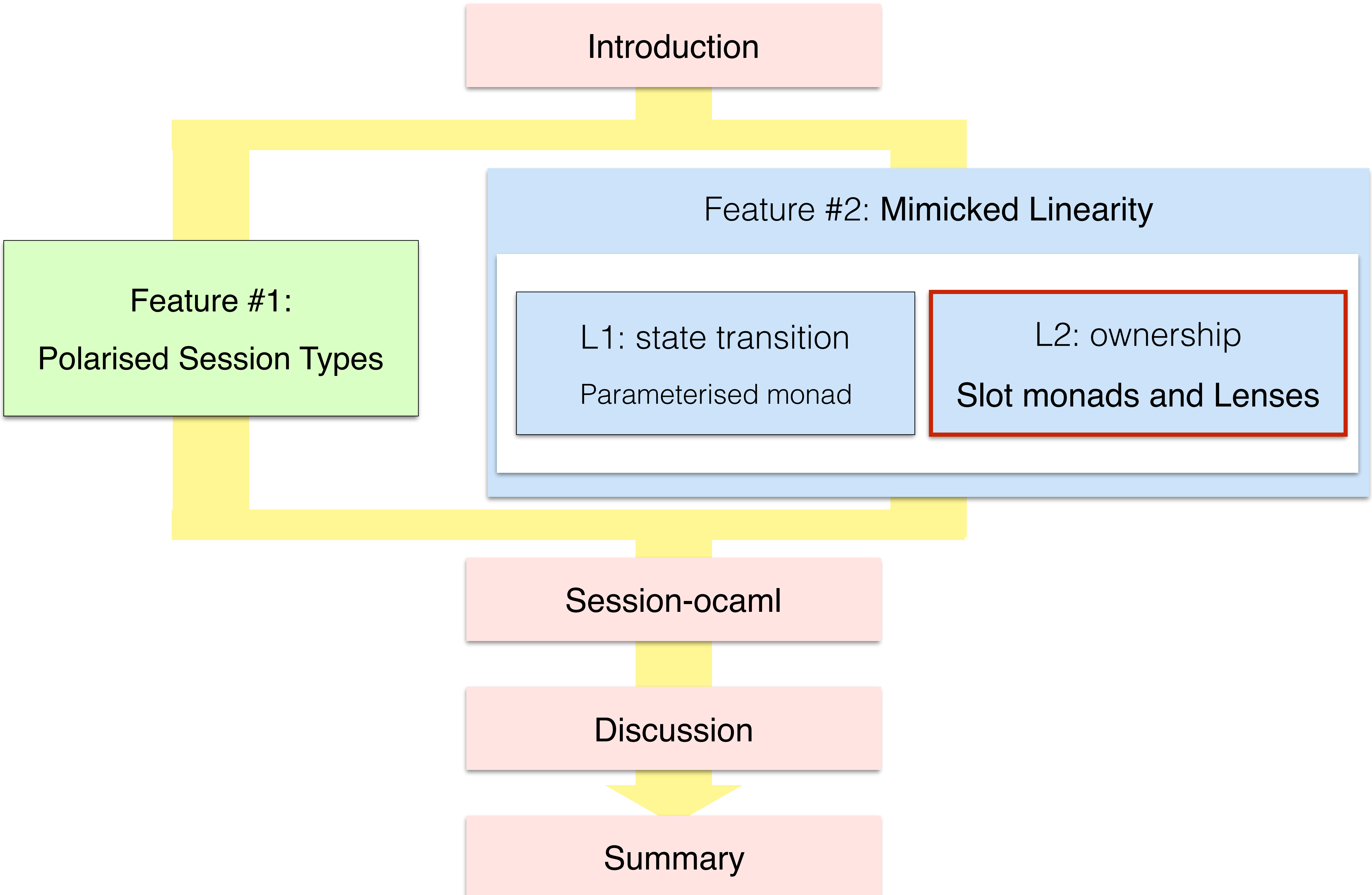



```
val close : (close, ε, unit) monad
```




```
... (* other primitives *) ...
```

Presentation structure



L2: Ownership and delegation

```
type (ρ1, ρ2, τ) monad
```

$$\rho1 \longrightarrow \rho2$$

... only tracks a single session.

Delegation involves **two sessions:**

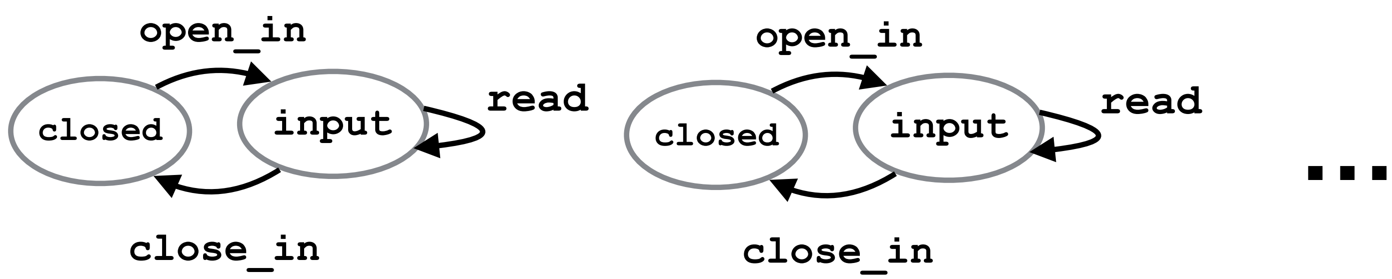
`delegate s0 t0`

`s0` $\quad !\theta;\sigma \longrightarrow \sigma$ $\quad$ progresses the state of **`s0`**, and

`t0` $\quad \theta \longrightarrow \varepsilon$ $\quad$ **releases the ownership** of **`t0`**

Garrigue's method (Safeio) ['06]: tracking multiple file handles

To track **multiple** file handles' states:

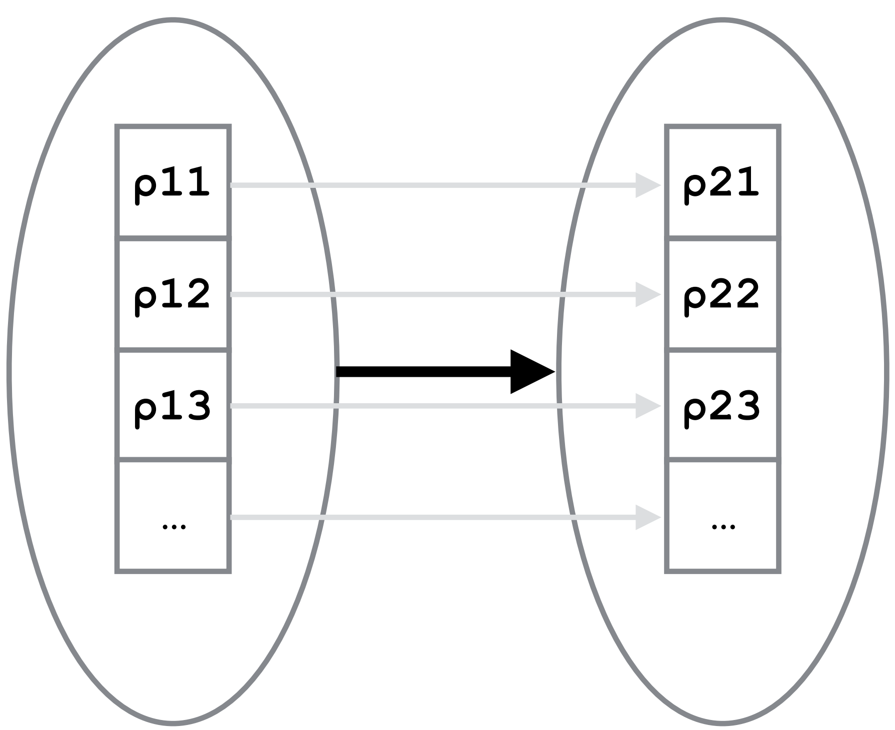


Embed vector of types (**slots**) in the parameterised monad (using cons-style):

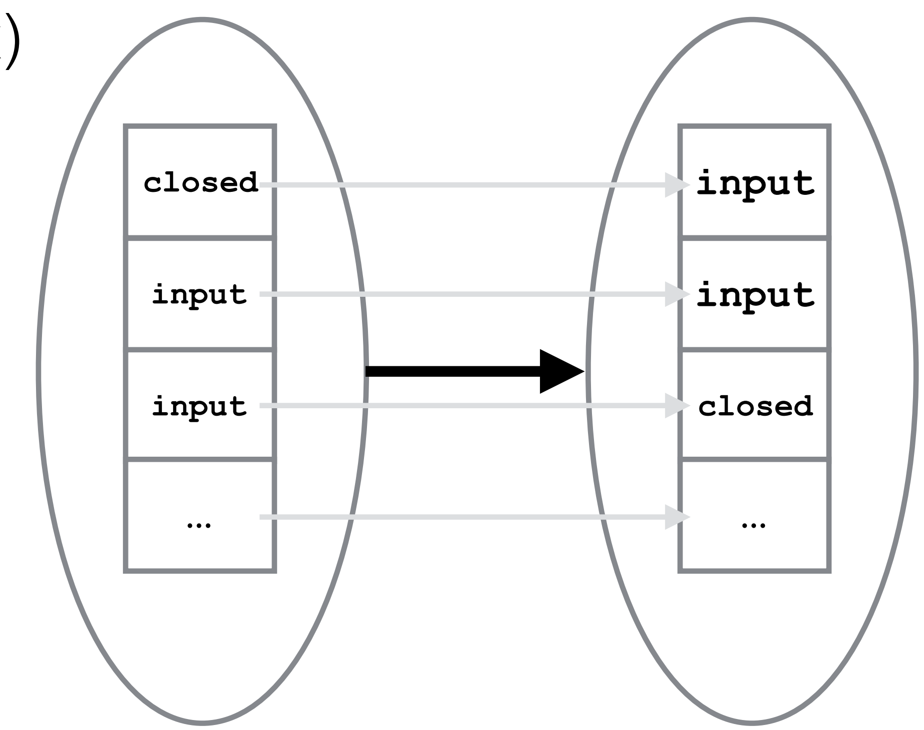
```
val a_file_op: ( ρ11*(ρ12*(ρ13*...)), ρ21*(ρ22*(ρ23*...)), τ) monad
```

Pictorially:

a_file_op:

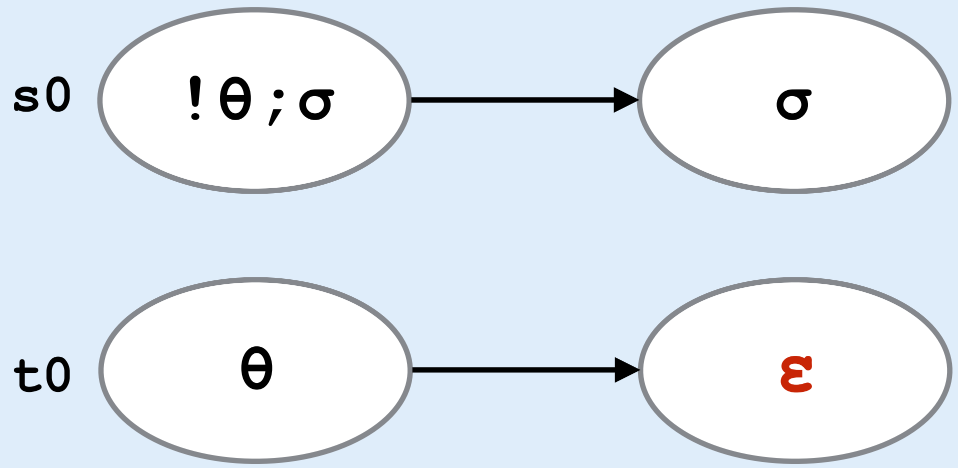


ex)



Solution to (L2): Lens to handle slots

`delegate s0 t0 :`

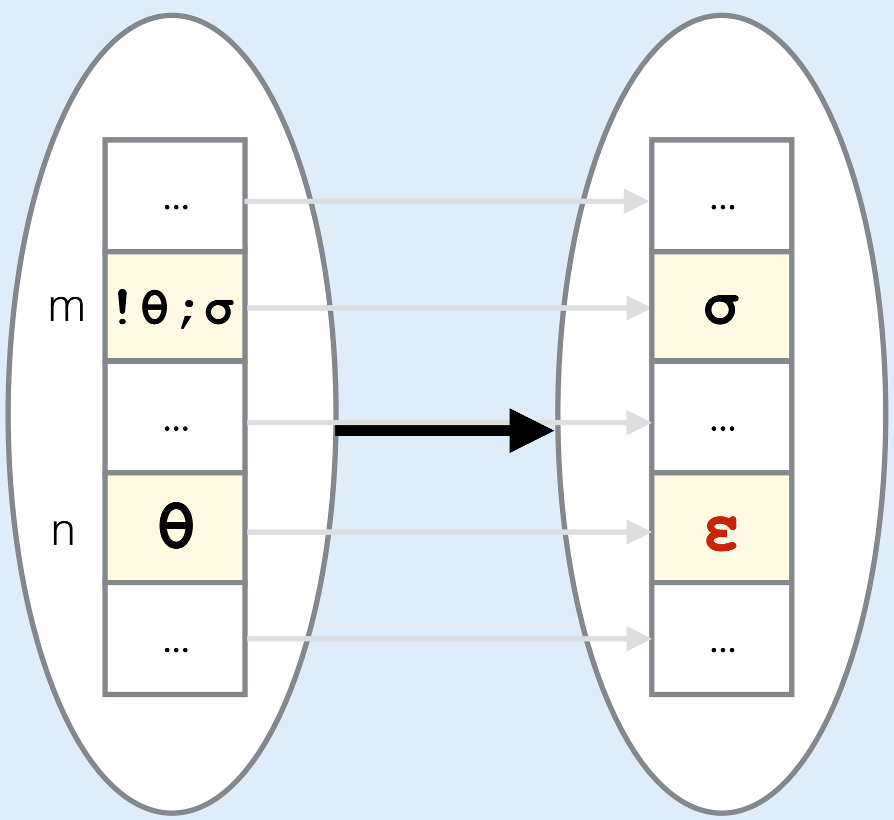


progresses the state of **`s0`**, and

releases the ownership of **`t0`**

Use **lenses** `_m`, `_n`, ... to specify the position *m*, *n*, ... in a vector:

`delegate _m _n :`



progresses the state of **`m`-th**, and

releases the ownership of **`n`-th**

... and keep the rest of slots untouched

Lenses [Foster et al.'05], [Pickering et al.'17]

```
type (θ1, θ2, ρ1, ρ2) lens
```

A lens is a function to **update the n-th element** of a type vector `ρ1` from `θ1` to `θ2`.

```
val _0: (θ1, θ2, θ1 * ρ, θ2 * ρ) lens
```


```
val _1: (θ1, θ2, ρ1 * (θ1 * ρ), ρ1 * (θ2 * ρ)) lens
```

See that the rest of vector remains unchanged.

Putting them altogether: Polarities and slots & lenses

```
let rec main () =
  accept eqch _0    >>
  connect wrkch _1 >>
  delegate _1 _0 >>=
  close _1
  main
```

```
let rec worker () =
  accept wrkch _0  >>
  deleg_recv _0 _1 >>
  close _0 >>
  match%branch _0 with
  | `bin -> let%s x,y = recv _0 in
            send _0 (x=y) >>=
            worker
  | `fin -> close _0
```

Putting them altogether: Polarities and slots & lenses

```
let rec main () =
  accept eqch _0    >>
  connect wrkch _1 >>
  delegate _1 _0 >>=
  close _1
  main
```

$0:\ \mathtt{req}[\theta^{\mathbf{serv}}];\mathtt{close}^{\mathbf{cli}}$

$1:\ \theta^{\mathbf{serv}} \to \varepsilon$

```
let rec worker () =
  accept wrkch _0  >>
  deleg_recv _0 _1 >>
  close _0 >>
  match%branch _0 with
  | `bin -> let%s x,y = recv _0 in
            send _0 (x=y) >>=
            worker
  | `fin -> close _0
```

$0:\ (\theta\mathtt{req}[\theta^{\mathbf{serv}}];\ \mathtt{close})^{\mathbf{serv}}$

$1:\ \varepsilon \to \theta^{\mathbf{serv}}$

polarised session types

$$1:\ \theta^{\mathbf{serv}} = \mu\alpha.\mathtt{req}\{\ \mathtt{bin}:\ \mathtt{req}[\mathtt{int*int}];\mathtt{resp}[\mathtt{bool}];\alpha,\ \mathtt{fin}:\ \mathtt{close}\ \}^{\mathbf{serv}}$$

Putting them altogether: Polarities and slots & lenses

```
let rec main () =
  accept eqch _0    >>
  connect wrkch _1 >>
  delegate _1 _0 >>=
  close _1
  main
```

$0:\ \mathtt{req}[\theta^{\mathtt{serv}}];\mathtt{close}^{\mathtt{cli}}$

$1:\ \theta^{\mathtt{serv}} \to \varepsilon$

```
let rec worker () =
  accept wrkch _0  >>
  deleg_recv _0 _1 >>
  close _0 >>
  match%branch _0 with
  | `bin -> let%s x,y = recv _0 in
              send _0 (x=y) >>=
              worker
  | `fin -> close _0
```

$0:\ (\theta\mathtt{req}[\theta^{\mathtt{serv}}];\ \mathtt{close})^{\mathtt{serv}}$

$1:\ \varepsilon \to \theta^{\mathtt{serv}}$

Lenses

polarised session types

$1:\ \theta^{\mathtt{serv}} = \mu\alpha.\mathtt{req}\{\ \mathtt{bin}:\ \mathtt{req}[\mathtt{int}*\mathtt{int}];\mathtt{resp}[\mathtt{bool}];\alpha,\ \mathtt{fin}:\ \mathtt{close}\ \}^{\mathtt{serv}}$

Putting them altogether: Polarities and slots & lenses

```
let rec main () =
  accept eqch _0    >>
  connect wrkch _1 >>
  delegate _1 _0 >>=
  close _1
  main
```

statically-typed delegation

$0:\ req[\theta^{serv}];close^{cli}$

$1:\ \theta^{serv} \to \varepsilon$

```
let rec worker () =
  accept wrkch _0  >>
  deleg_recv _0 _1 >>
  close _0 >>
  match%branch _0 with
  | `bin -> let%s x,y = recv _0 in
            send _0 (x=y) >>=
            worker
  | `fin -> close _0
```

$0:\ (\theta req[\theta^{serv}];\ close)^{serv}$

$1:\ \varepsilon \to \theta^{serv}$

Lenses

polarised session types

$1:\ \theta^{serv} = \mu\alpha.req\{\ bin:\ req[int*int];resp[bool];\alpha,\ fin:\ close\ \}^{serv}$

Putting them altogether: Polarities and slots & lenses

```
let rec main () =
  accept eqch _0    >>
  connect wrkch _1  >>
  delegate _1 _0 >>=
  close _1
  main
```

0: $req[\theta^{serv}]; close^{cli}$

1: $\theta^{serv} \rightarrow \varepsilon$

statically-typed **delegation**

```
let rec worker () =
  accept wrkch _0  >>
  deleg_recv _0 _1 >>
  close _0 >>
  match%branch _0 with
  | `bin -> let%s x,y = recv _0 in
            send _0 (x=y) >>=
            worker
  | `fin -> close _0
```

0: $(\theta req[\theta^{serv}]; close)^{serv}$

1: $\varepsilon \rightarrow \theta^{serv}$

Lenses

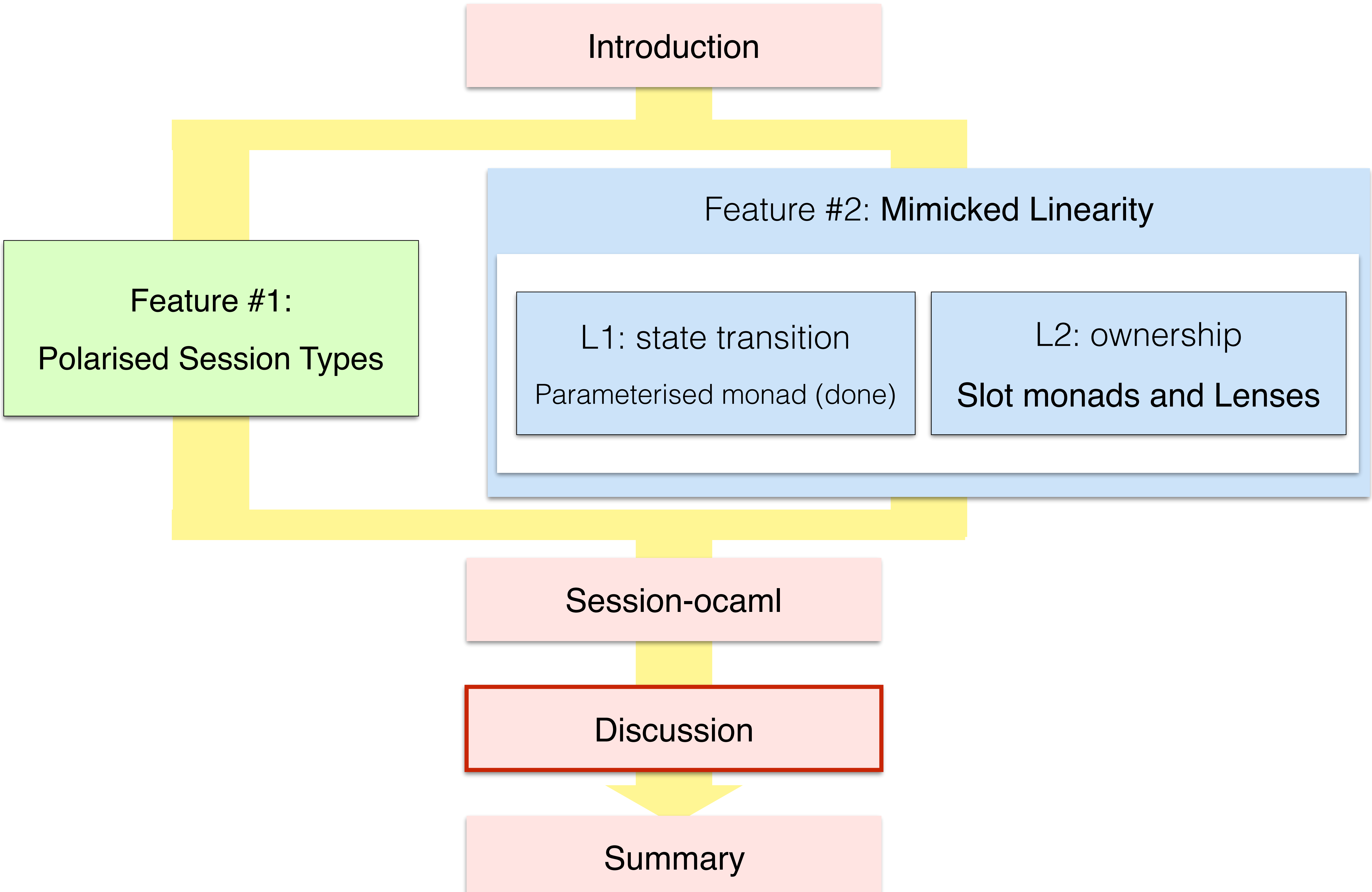
polarised session types

1: $\theta^{serv} = \mu\alpha.req\{ bin: req[int*int]; resp[bool]; \alpha, fin: close \}^{serv}$

session-type inference

```
val eqch  : μα.req{ bin: req[int*int];resp[bool];α,
                    fin: close }
val wrkch : req[ ... ];close
```

Presentation structure



Comparing OCaml implementations

L1) State transition in types

L2) Tracking ownership of a session endpoint

	L1	L2	Static/Dynamic	Duality Infer.
Imai et al.	✔	✔	**static**	**Polarised**
Padovani (1)	✔	✔	**dynamic**	Dardha's encoding
Padovani (2)	✔	×	static	Dardha's encoding
Pucella & Tov	✔	×	static	Manual

OCaml v.s. Haskell; implementing languages

- **OCaml** implementation results in **simpler** one
  - Only use **parametric polymorphism**
  - Exportable to other languages
  - Slight **notational overhead** to use slots (_0, _1, ...)

```
('s,'t,'p,'q) slot -> ... ->
('p,'q,'a) monad
```

[Imai, Yoshida & Yuen, '17]

- **Portable** to other functional languages (Standard ML) or even other non-FP languages

- **Haskell** uses much **complex type-features**
  - '**Complex**' features like type functions, functional dependencies, higher-order types and so on.

```
(Pickup ss n s,
 Update ss n t ss',
 IsEnded ss f) =>
... -> Session t ss ss' ()
```

[Imai et al., '10]

```
(GV ch repr, DualSession s) =>
... -> repr v i o (ch s)
```

[Lindley & Morris, '16]

  - More **natural and idiomatic** to use

The paper includes

- Details of **lens-typed** communication primitives
- Examples
  - Travel agency [Hu et al, 2008]
    with delegation and make use of type inference
  - SMTP client (Session-typed SMTP protocol)
    Practical network programing, no delegation
  - A database server
    With delegation
  - Session-ocaml clearly describes these examples!

Summary

- **Session-ocaml**: a full-fledged session type implementation in OCaml
    - ***Polarised session types***
    - ***Slot monad*** **and** ***lenses -- Linearity!***

Available at: **`https://github.com/keigoi/session-ocaml/`**

- Session-ocaml is a simple yet powerful playground for session-typed programming
- Further work:

  Extension to multiparty session types, Java and C# implementation,

  and so on

- Supplemental slides

Dynamic checking on Linearity

- Trying to send `"*"` repeatedly in FuSe [Padovani '16], but fails:

```
let rec loop () =
  let s' = send "*" s
  in
  match branch s' with
  | `stop s'' -> close s''
  | `cont _ -> loop ()
```

- Session-ocaml's Slot-Oriented Programming offers a **statically-checked** alternative.

Dynamic checking on Linearity

- Trying to send `"*"` repeatedly in FuSe [Padovani '16], but fails:

```
let rec loop () =
  let s' = send "*" s
  in
  match branch s' with
  | `stop s'' -> close s''
  | `cont _ -> loop ()
```

discarding the new session endpoint

- Session-ocaml's Slot-Oriented Programming offers a **statically-checked** alternative.

Dynamic checking on Linearity

- Trying to send "*" repeatedly in FuSe [Padovani '16], but fails:

runtime-error on second iteration

```
let rec loop () =
  let s' = send "*" s
  in
  match branch s' with
  | `stop s'' -> close s''
  | `cont _ -> loop ()
```

discarding the new session endpoint

- Session-ocaml's Slot-Oriented Programming offers a **statically-checked** alternative.

Dynamic checking on Linearity

- Trying to send "*" repeatedly in FuSe [Padovani '16], but fails:

```
let rec loop () =
  let s' = send "*" s
  in
  match branch s' with
  | `stop s'' -> close s''
  | `cont _ -> loop ()
```

runtime-error on second iteration

discarding the new session endpoint

Correct version:

```
let rec loop s =
  let s' = send "*" s
  in
  match branch s' with
  | `stop s'' -> close s''
  | `cont s'' -> loop s''
```

- Session-ocaml's Slot-Oriented Programming offers a **statically-checked** alternative.

Two versions of Session-ocaml: `Session0` and `SessionN`

`module Session0` (single-session)		`module SessionN` (multiple-sessions)
`accept_ ch (fun () -> ...)` `connect_ ch (fun () -> ...)`	establishing a session	`accept ch ~bindto:_n` (slot specifier (lens)) `connect ch ~bindto:_n`
`send "Hello"`	sending a value	`send _n "Hello"`
`let%s x = recv () in`	receive a value	`let%s x = recv _n in ...`
``[%select0 `Apple]``	label selection	``[%select _n `Apple]``

``match%branch0 () with``
``| `Apple -> ...``
``| `Banana -> ...``
	labelled branching	``match%branch _n with``
		``| `Apple -> ...``
		``| `Banana -> ...``

	delegation	`deleg_send _n ~release:_m` (delegation supported)
	accepting delegation	`deleg_recv _n ~bindto:_m`