

FuSe: An Ocaml implementation of binary session types¹

¹Padovani, L. (2017). A simple library implementation of binary sessions. Journal of Functional Programming, 27. The implementation can be downloaded from <https://github.com/boystrange/FuSe>

Session Types

Syntax

$$\begin{aligned} t, s &::= \text{bool} \mid \text{int} \mid \dots \mid T \mid \alpha \mid [l_i : t_i]_{i \in I} \\ T, S &::= \text{end} \mid !t.T \mid ?t.T \mid \oplus[l_i : T_i]_{i \in I} \mid \&[l_i : T_i]_{i \in I} \mid A \mid \bar{A} \end{aligned}$$

- ▶ FuSe provides polymorphic session types
- ▶ α is a type variable

? α .! α .end: a session type for an endpoint that starts by receiving a value of some type α (e.g., any type) and then sends back a value of the same type

- ▶ A is a session type variable

?int. A : a session type for an endpoint that starts by receiving an integer value and then follows by a session type (e.g., any session type)

- ▶ \bar{A} the dual of a session type variable

Duality

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

$$\overline{(?t.T)} = !t.\overline{T}$$

$$\overline{(!t.T)} = ?t.\overline{T}$$

$$\overline{\&[1_i : T_i]_{i \in I}} = \oplus[1_i : \overline{T}_i]_{i \in I}$$

$$\overline{\oplus[1_i : T_i]_{i \in I}} = \&[1_i : \overline{T}_i]_{i \in I}$$

$$\overline{\overline{A}} = A$$

An API for sessions

Module Session

```
val send      : α → !α.A → A
val receive   : ?α.A → α × A
val close     : end → unit
val create    : unit → A ×  $\overline{A}$ 
```

Echo client

```
let echo_client ep x =
  let ep = Session.send x ep in
  let res, ep = Session.receive ep in
  Session.close ep;
  res
```

```
echo_client : !α.?β.end → α → β
```

Echo service

```
let echo_service ep =
  let x, ep = Session.receive ep in
  let ep = Session.send x ep in
  Session.close ep

echo_service : ?α.!α.end → unit
```

Duality and parametric polymorphism

```
echo_client  : !α.?β.end → α → β
echo_service : ?α.!α.end → unit
```

Note that:

$$\overline{!α.?β.end} = ?α.!β.end \neq ?α.!α.end$$

However

- ▶ $?α.!β.end$ is more general than $?α.!α.end$
 - ▶ Recall that $?α.!β.end$ stands for $\forall α. \forall β. ?α.!β.end$
- ▶ $\forall α.?α.!α.end$ is a particular instance
- ▶ there is a unification for $?α.!β.end$ and $?α.!α.end$

Session creation

```
let _ =
  let a, b = Session.create () in
  let _ = Thread.create echo_service a in
  print_endline (echo_client b "Hello, world!")
```

Session Types

Syntax

$$\begin{aligned} t, s ::= & \text{ bool } | \text{ int } | \cdots | T | \alpha | [l_i : t_i]_{i \in I} \\ T, S ::= & \text{ end } | !t.T | ?t.T | \&[l_i : T_i]_{i \in I} | \oplus[l_i : T_i]_{i \in I} | A | \overline{A} \end{aligned}$$

- ▶ $[l_i : t_i]_{i \in I}$: Variants (disjoint sums)

Variants in Ocaml

```
type role = Student | Teacher

let role_to_string r =
  match r with
  | Student → "Student"
  | Teacher → "Teacher"

let _ =
  print_string (role_to_string Student)
```

Variants in Ocaml

```
type role = Student of string | Teacher of int

let role_to_string r =
  match r with
  | Student name → "Student " ^ name
  | Teacher id → "Teacher " ^ (string_of_int id)

let _ =
  print_string (role_to_string (Student "Alice"))
```

```
type role = Student | Teacher
val role_to_string : role → string
```

Polymorphic Variants in Ocaml

- ▶ Limitation of (ordinary) variants: Labels (or constructors) are limited to those declared by the type
- ▶ We need the flexibility of choosing the set of labels (each protocol needs its own labels)
- ▶ Solution: Polymorphic Variants

```
let role_to_string r =
  match r with
    | `Student name → "Student " ^ name
    | `Teacher id → "Teacher " ^ (string_of_int id)
```

```
let _ =
  print_string (role_to_string (`Student "Alice"))
```

```
val role_to_string : [< `Student of string | `Teacher of int ]
                      → string
```

An API for sessions

Module Session

```
val send      : α → !α.A → A
val receive   : ?α.A → α × A
val create    : unit → A ×  $\overline{A}$ 
val close     : end → unit
val branch   : &[li : Ai]i ∈ I → [li : Ai]i ∈ I
```

Branch

```
echo_service : ?α.!α.end → unit
```

```
val branch : &[li : Ai]i ∈ I → [li : Ai]i ∈ I
```

```
val opt_echo_service : &[End : end,Msg : ?α.!α.end] → unit
```

```
let opt_echo_service ep =
  match Session.branch ep with
  | `Msg ep → echo_service ep
  | `End ep → Session.close ep
```

An API for sessions

Module Session

```
val send      : α → !α.A → A
val receive   : ?α.A → α × A
val create    : unit → A ×  $\overline{A}$ 
val close     : end → unit
val branch   : &[li : Ai]i ∈ I → [li : Ai]i ∈ I
val select   : ( $\overline{A}_k$  → [li :  $\overline{A}_i$ ]i ∈ I) →  $\oplus$ [li : Ai]i ∈ I → Ak
```

Select

```
val select : ( $\overline{A_k} \rightarrow [l_i : \overline{A_i}]_{i \in I}$ ) →  $\oplus[l_i : A_i]_{i \in I} \rightarrow A_k$ 
```

```
opt_echo_client :  $\Theta[\text{End} : \text{end}, \text{Msg} : !\alpha.\alpha.\text{end}] \rightarrow \text{bool} \rightarrow \alpha \rightarrow \alpha$ 
```

```
let opt_echo_client ep opt x =
  if opt then
    let ep = Session.select (fun y → `Msg y) ep in
    let ep = Session.send x ep in
    let reply, ep = Session.receive ep in
    Session.close ep;
    reply
  else
    let ep = Session.select (fun y → `End y) ep in
    Session.close ep; x
```

Subtyping

Thanks to polymorphic variants, the implementation allows for subtyping:

```
let end_echo_client ep =
  let ep = Session.select (fun x → `End x) ep
  in Session.close ep
```

```
val end_echo_client: Φ[End : end] → unit
```

```
val opt_echo_service : &[End : end,Msg : ?α.!α.end] → unit
```

Note that:

$$\Phi[\text{End} : \text{end}] = \&[\text{End} : \text{end}] \neq \&[\text{End} : \text{end}, \text{Msg} : ?\alpha.! \alpha.\text{end}]$$

This is handled by a notion of subtyping (or safe substitution)

Subtyping

For this reason the following code is well-typed

```
val end_echo_client: Θ[End : end] → unit
```

```
val opt_echo_service : &[End : end,Msg : ?α..!α.end] → unit
```

```
let _ =
let a, b = Session.create () in
let _ = Thread.create opt_echo_service a in
end_echo_client b
```

Recursive types

```
let rec rec_echo_service ep =
  match Session.branch ep with
  | `Msg ep → let x, ep = Session.receive ep in
    let ep = Session.send x ep in
    rec_echo_service ep
  | `End ep → Session.close ep
```

```
val rec_echo_service : rec A.&[End : end,Msg : ?α. !α.A] → unit
```

`rec A.T` denotes the (equi-recursive) session type T in which occurrences of A stand for the session type itself.

Recursive types

```
let rec rec_echo_client ep =
  function
  | [] → let ep = Session.select _End ep in
    Session.close ep; []
  | x :: xs → let ep = Session.select _Msg ep in
    let ep = Session.send x ep in
    let y, ep = Session.receive ep in
    y :: rec_echo_client ep xs
```

```
val rec_echo_client : rec A.θ[End : end, Msg : !α.?β.A]
                      → α list → β list
```

rec $A.T$ denotes the (equi-recursive) session type T in which occurrences of A stand for the session type itself.

Recursive types and Subtyping

```
let rec_echo_client_2 ep x =  
  let ep = Session.select (fun x → `Msg x) ep in  
  let ep = Session.send x ep in  
  let res, ep = Session.receive ep in  
  let ep = Session.select (fun x → `End x) ep in  
  Session.close ep;  
  res
```

```
val rec_echo_client_2 : ∀[Msg : !α. ?β. ∀[End : end]] → α → β
```

This case also holds by subtyping

Implementation: Representation of types

Main idea

- ▶ Session types: Products + Sums + Linearity
- ▶ Ornella Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. PPDP'12.

Two types

- ▶ \emptyset , which is not inhabited (no constructor)
- ▶ $\langle \rho, \sigma \rangle$ for channels:
 - ▶ receiving messages of type ρ
 - ▶ sending messages of type σ .
 - ▶ ρ and σ instantiated with \emptyset to indicate that no message is respectively received and/or sent

Representation of session types

Encoding

$$\begin{aligned}\llbracket \text{end} \rrbracket &= \langle \emptyset, \emptyset \rangle \\ \llbracket ?t . T \rrbracket &= \langle \llbracket t \rrbracket \times \llbracket T \rrbracket, \emptyset \rangle \\ \llbracket !t . T \rrbracket &= \langle \emptyset, \llbracket t \rrbracket \times \llbracket \bar{T} \rrbracket \rangle \\ \llbracket \&[\mathbf{l}_i : T_i]_{i \in I} \rrbracket &= \langle [\mathbf{l}_i : \llbracket T_i \rrbracket]_{i \in I}, \emptyset \rangle \\ \llbracket \oplus[\mathbf{l}_i : T_i]_{i \in I} \rrbracket &= \langle \emptyset, [\mathbf{l}_i : \llbracket \bar{T}_i \rrbracket]_{i \in I} \rangle \\ \llbracket A \rrbracket &= \langle \rho_A, \sigma_A \rangle \\ \llbracket \bar{A} \rrbracket &= \langle \sigma_A, \rho_A \rangle\end{aligned}$$

Examples

? α . A

$$[\![? \alpha . A]\!] = \langle \alpha \times \langle \rho_A, \sigma_A \rangle, \emptyset \rangle$$

$T = \oplus[\text{End} : \text{end}, \text{Msg} : !\alpha . ?\beta . \text{end}]$

$$\begin{aligned} [\![T]\!] &= \langle \emptyset, [\text{End} : [\![\text{end}]\!], \text{Msg} : [\![? \alpha . ! \beta . \text{end}]\!]] \rangle \\ &= \langle \emptyset, [\text{End} : \langle \emptyset, \emptyset \rangle, \text{Msg} : \langle \alpha \times [\![! \beta . \text{end}]\!], \emptyset \rangle] \rangle \\ &= \langle \emptyset, [\text{End} : \langle \emptyset, \emptyset \rangle, \text{Msg} : \langle \alpha \times \langle \emptyset, \beta \times [\![\text{end}]\!], \emptyset \rangle] \rangle \\ &= \langle \emptyset, [\text{End} : \langle \emptyset, \emptyset \rangle, \text{Msg} : \langle \alpha \times \langle \emptyset, \beta \times \langle \emptyset, \emptyset \rangle \rangle, \emptyset \rangle] \rangle \end{aligned}$$

$\bar{T} = \&[\text{End} : \text{end}, \text{Msg} : ?\alpha . !\beta . \text{end}]$

$$\begin{aligned} [\![\bar{T}]\!] &= \langle [\text{End} : [\![\text{end}]\!], \text{Msg} : [\![? \alpha . ! \beta . \text{end}]\!]], \emptyset \rangle \\ &= \langle [\text{End} : \langle \emptyset, \emptyset \rangle, \text{Msg} : \langle \alpha \times [\![! \beta . \text{end}]\!], \emptyset \rangle], \emptyset \rangle \\ &= \langle [\text{End} : \langle \emptyset, \emptyset \rangle, \text{Msg} : \langle \alpha \times \langle \emptyset, \beta \times [\![\text{end}]\!], \emptyset \rangle], \emptyset \rangle \\ &= \langle [\text{End} : \langle \emptyset, \emptyset \rangle, \text{Msg} : \langle \alpha \times \langle \emptyset, \beta \times \langle \emptyset, \emptyset \rangle \rangle, \emptyset \rangle], \emptyset \rangle \end{aligned}$$

Representation of session types

Theorem

If $\llbracket T \rrbracket = \langle t, s \rangle$, then $\llbracket \bar{T} \rrbracket = \langle s, t \rangle$.

Interface in Ocaml

Session

```
module Session : sig
  type ⊕
  type (ρ,σ) st (* OCaml syntax for <ρ,σ> *)
  val create   : unit → (ρ,σ) st × (σ,ρ) st
  val close    : (⊖,⊖) st → unit
  val send     : α → (⊖,(α × (σ,ρ) st)) st → (ρ,σ) st
  val receive  : ((α × (ρ,σ) st),⊖) st → α × (ρ,σ) st
  val select   : ((σ,ρ) st → α) → (⊖,[>] as α) st → (ρ,σ) st
  val branch   : ([>] as α,⊖) st → α
end
```

Implementation of session types

Untyped channels

```
module UnsafeChannel : sig
  type t
  val create    : unit → t
  val send      : α → t → unit
  val receive   : t → α
end
```

`UnsafeChannel` is implemented on top of `Event.channel`.

Implementation of session types

Untyped channels

```
module UnsafeChannel : sig
  type t
  val create    : unit → t
  val send      : α → t → unit
  val receive   : t → α
end
```

`UnsafeChannel` is implemented on top of `Event.channel`.

Implementation of session types

```
type ( $\alpha, \beta$ ) st = { chan : UnsafeChannel.t;  
                      mutable valid : bool }
```

- ▶ valid is used for run-time checking of linearity

Implementation of session types

```
val create : unit → (ρ,σ) st × (σ,ρ) st
```

```
let create () = let ch = UnsafeChannel.create ()
               in { chan = ch; valid = true },
                  { chan = ch; valid = true }
```

Implementation of session types

```
val close    : (Ø, Ø) st → unit  
  
let close = use  
  
let use u = if u.valid then u.valid ← false  
            else raise InvalidEndpoint
```

Implementation of session types

```
val send      : α → ((),(α × (σ,ρ) st)) st → (ρ,σ) st
val receive   : ((α × (ρ,σ) st),()) st → α × (ρ,σ) st
```

```
let send x u =
  use u; UnsafeChannel.send x u.chan; fresh u
let receive u =
  use u; (UnsafeChannel.receive u.chan, fresh u)
```

```
let fresh u = { u with valid = true }
```

Implementation of session types

```
val select  : ((σ,ρ) st → α) → (Ø,[>] as α) st → (ρ,σ) st
val branch  : ([>] as α,Ø) st → α
```

```
let select = send
let branch u =
  use u; UnsafeChannel.receive u.chan (fresh u)
```

Actual types inferred by Ocaml

```
val rec_echo_client :  
  ((),[> `End of ((),()) st  
  | `Msg of (beta * ((),gamma * ((),alpha) st) st,()) st]  
    as alpha) st → beta list → gamma list
```

The session type

```
val rec_echo_client :  
  rec X.θ[ End: end | Msg: !alpha.?beta.X ] →  
  alpha list → beta list
```

is obtained by encoding back the representation²

²pretty printing is performed by rosetta tool

Non linear usage of channels

```
let client ep x y =
  let _ = Session.send x ep in
  let ep = Session.send y ep in
  let result, ep = Session.receive ep in
  Session.close ep;
  result

let service ep =
  let x, ep = Session.receive ep in
  let ep = Session.send x ep in
  Session.close ep

let _ =
  let a, b = Session.create () in
  let _ = Thread.create service a in
  print_int (client b 1 2)
```

The program is well-typed

```
val client : !α.?α. → α → α → β
val service : ?α.!β. → unit
```

Its execution raises the exception `Session.InvalidEndpoint`