

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

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

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

$? \alpha . ! \alpha . \text{end}$: 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

$? \text{int} . A$: 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      :  $\alpha \rightarrow !\alpha.A \rightarrow A$   
val receive   :  $? \alpha.A \rightarrow \alpha \times A$   
val close     : end  $\rightarrow$  unit  
val create    : unit  $\rightarrow A \times \overline{A}$ 
```

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 : ! $\alpha$ .? $\beta$ .end  $\rightarrow$   $\alpha \rightarrow \beta$ 
```

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:

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

However

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

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

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

- ▶ $[l_i : t_i]_{i \in I}$: 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      :  $\alpha \rightarrow !\alpha.A \rightarrow A$   
val receive   :  $? \alpha.A \rightarrow \alpha \times A$   
val create    : unit  $\rightarrow A \times \overline{A}$   
val close     : end  $\rightarrow$  unit  
val branch    :  $\&[1_i : A_i]_{i \in I} \rightarrow [1_i : A_i]_{i \in I}$ 
```

Branch

```
echo_service : ? $\alpha$ .! $\alpha$ .end  $\rightarrow$  unit
```

```
val branch : &[ $l_i : A_i$ ] $_{i \in I} \rightarrow [l_i : A_i]_{i \in I}$ 
```

```
val opt_echo_service : &[End : end, Msg : ? $\alpha$ .! $\alpha$ .end]  $\rightarrow$  unit
```

```
let opt_echo_service ep =  
  match Session.branch ep with  
  | `Msg ep  $\rightarrow$  echo_service ep  
  | `End ep  $\rightarrow$  Session.close ep
```

An API for sessions

Module Session

```
val send      :  $\alpha \rightarrow !\alpha.A \rightarrow A$   
val receive   :  $?\alpha.A \rightarrow \alpha \times \overline{A}$   
val create    :  $\text{unit} \rightarrow A \times \overline{A}$   
val close     :  $\text{end} \rightarrow \text{unit}$   
val branch    :  $\&[\underline{1}_i : A_i]_{i \in I} \rightarrow [\underline{1}_i : A_i]_{i \in I}$   
val select    :  $(\overline{A}_k \rightarrow [\underline{1}_i : \overline{A}_i]_{i \in I}) \rightarrow \oplus[\underline{1}_i : A_i]_{i \in I} \rightarrow A_k$ 
```

Select

```
val select :  $(\overline{A_k} \rightarrow [1_i : \overline{A_i}]_{i \in I}) \rightarrow \oplus [1_i : A_i]_{i \in I} \rightarrow A_k$ 
```

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

```
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:  ⊕[End : end] → unit
```

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

Note that:

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

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:  $\oplus$ [End : end]  $\rightarrow$  unit
```

```
val opt_echo_service :  $\&$ [End : end, Msg : ? $\alpha$ .! $\alpha$ .end]  $\rightarrow$  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.⊕[End: end,Msg: !α.?β.A]  
                  → α list → β list
```

$\text{rec } A.T$ 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 :  $\oplus[\text{Msg} : !\alpha.?\beta.\oplus[\text{End} : \text{end}]] \rightarrow \alpha \rightarrow \beta$ 
```

This case also holds by subtyping

Implementation: Representation of types

Main idea

- ▶ Session types: Products + Sums + Linearity
- ▶ Ornela 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

$$\begin{aligned}\llbracket \text{end} \rrbracket &= \langle \emptyset, \emptyset \rangle \\ \llbracket ?t. T \rrbracket &= \langle \llbracket t \rrbracket \times \llbracket T \rrbracket, \emptyset \rangle \\ \llbracket !t. T \rrbracket &= \langle \emptyset, \llbracket t \rrbracket \times \llbracket \overline{T} \rrbracket \rangle \\ \llbracket \&[1_i : T_i]_{i \in I} \rrbracket &= \langle [1_i : \llbracket T_i \rrbracket]_{i \in I}, \emptyset \rangle \\ \llbracket \oplus[1_i : T_i]_{i \in I} \rrbracket &= \langle \emptyset, [1_i : \llbracket \overline{T}_i \rrbracket]_{i \in I} \rangle \\ \llbracket A \rrbracket &= \langle \rho_A, \sigma_A \rangle \\ \llbracket \overline{A} \rrbracket &= \langle \sigma_A, \rho_A \rangle\end{aligned}$$

Examples

$? \alpha . A$

$$\llbracket ? \alpha . A \rrbracket = \langle \alpha \times \langle \rho_A, \sigma_A \rangle, 0 \rangle$$

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

$$\begin{aligned} \llbracket T \rrbracket &= \langle 0, [\text{End} : \llbracket \text{end} \rrbracket, \text{Msg} : \llbracket ? \alpha . ! \beta . \text{end} \rrbracket] \rangle \\ &= \langle 0, [\text{End} : \langle 0, 0 \rangle, \text{Msg} : \langle \alpha \times \llbracket ! \beta . \text{end} \rrbracket, 0 \rangle] \rangle \\ &= \langle 0, [\text{End} : \langle 0, 0 \rangle, \text{Msg} : \langle \alpha \times \langle 0, \beta \times \llbracket \text{end} \rrbracket \rangle, 0 \rangle] \rangle \\ &= \langle 0, [\text{End} : \langle 0, 0 \rangle, \text{Msg} : \langle \alpha \times \langle 0, \beta \times \langle 0, 0 \rangle \rangle, 0 \rangle] \rangle \end{aligned}$$

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

$$\begin{aligned} \llbracket \bar{T} \rrbracket &= \langle [\text{End} : \llbracket \text{end} \rrbracket, \text{Msg} : \llbracket ? \alpha . ! \beta . \text{end} \rrbracket], 0 \rangle \\ &= \langle [\text{End} : \langle 0, 0 \rangle, \text{Msg} : \langle \alpha \times \llbracket ! \beta . \text{end} \rrbracket, 0 \rangle], 0 \rangle \\ &= \langle [\text{End} : \langle 0, 0 \rangle, \text{Msg} : \langle \alpha \times \langle 0, \beta \times \llbracket \text{end} \rrbracket \rangle, 0 \rangle], 0 \rangle \\ &= \langle [\text{End} : \langle 0, 0 \rangle, \text{Msg} : \langle \alpha \times \langle 0, \beta \times \langle 0, 0 \rangle \rangle, 0 \rangle], 0 \rangle \end{aligned}$$

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 0
  type ( $\rho, \sigma$ ) st (* OCaml syntax for  $\langle \rho, \sigma \rangle$  *)
  val create : unit  $\rightarrow$  ( $\rho, \sigma$ ) st  $\times$  ( $\sigma, \rho$ ) st
  val close : ( $0, 0$ ) st  $\rightarrow$  unit
  val send :  $\alpha \rightarrow$  ( $0, (\alpha \times (\sigma, \rho) \text{ st})$ ) st  $\rightarrow$  ( $\rho, \sigma$ ) st
  val receive : (( $\alpha \times (\rho, \sigma) \text{ st}$ ),  $0$ ) st  $\rightarrow$   $\alpha \times (\rho, \sigma) \text{ st}$ 
  val select : (( $\sigma, \rho$ ) st  $\rightarrow$   $\alpha$ )  $\rightarrow$  ( $0, [ > ]$  as  $\alpha$ ) st  $\rightarrow$  ( $\rho, \sigma$ ) st
  val branch : ([ > ] as  $\alpha, 0$ ) st  $\rightarrow$   $\alpha$ 
end
```

Implementation of session types

Untyped channels

```
module UnsafeChannel : sig
  type t
  val create      : unit → t
  val send        :  $\alpha$  → t → unit
  val receive     : t →  $\alpha$ 
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        :  $\alpha$  → t → unit
  val receive     : t →  $\alpha$ 
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 → ( $\rho, \sigma$ ) st × ( $\sigma, \rho$ ) st
```

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

Implementation of session types

```
val close    : ( $\emptyset$ , $\emptyset$ ) st  $\rightarrow$  unit
```

```
let close = use
```

```
let use u = if u.valid then u.valid  $\leftarrow$  false  
            else raise InvalidEndpoint
```

Implementation of session types

```
val send      :  $\alpha \rightarrow (\emptyset, (\alpha \times (\sigma, \rho) \text{ st})) \text{ st} \rightarrow (\rho, \sigma) \text{ st}$   
val receive   :  $((\alpha \times (\rho, \sigma) \text{ st}), \emptyset) \text{ st} \rightarrow \alpha \times (\rho, \sigma) \text{ st}$ 
```

```
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  : (( $\sigma, \rho$ ) st  $\rightarrow \alpha$ )  $\rightarrow$  ( $\emptyset, [ > ]$  as  $\alpha$ ) st  $\rightarrow$  ( $\rho, \sigma$ ) st  
val branch : ([ > ] as  $\alpha, \emptyset$ ) st  $\rightarrow \alpha$ 
```

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

Actual types inferred by Ocaml

```
val rec_echo_client :  
  (0,[> `End of (0,0) st  
    | `Msg of ( $\beta \times (0, \gamma \times (0, \alpha) \text{ st}) \text{ st}, 0) \text{ st}]$   
    as  $\alpha$ ) st  $\rightarrow \beta \text{ list} \rightarrow \gamma \text{ list}$ 
```

The session type

```
val rec_echo_client :  
  rec X. $\oplus$ [ End: end | Msg: ! $\alpha. ?\beta.X$  ]  $\rightarrow$   
   $\alpha \text{ list} \rightarrow \beta \text{ list}$ 
```

is obtained by encoding back the representation²

²pretty printing is preformed 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 : ! $\alpha$ .? $\alpha$ .  $\rightarrow \alpha \rightarrow \alpha \rightarrow \beta$   
val service : ? $\alpha$ .! $\beta$ .  $\rightarrow$  unit
```

Its execution raises the exception `Session.InvalidEndpoint`