`FuSe`: An Ocaml implementation of binary session types[1]

Hernán Melgratti

ICC University of Buenos Aires-Conicet

14 February 2020 @ Pisa

[1] Padovani, L. (2017). A simple library implementation of binary sessions. Journal of Functional Programming, 27.

Session Types

Syntax

$$t, s ::= \texttt{bool} \mid \texttt{int} \mid \cdots \mid T \mid \alpha \mid [\texttt{l}_i : t_i]_{i \in I}$$
$$T, S ::= \texttt{end} \mid !t.T \mid ?t.T \mid \oplus[\texttt{l}_i : T_i]_{i \in I} \mid \&[\texttt{l}_i : T_i]_{i \in I} \mid A \mid \overline{A}$$

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

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

 $?\texttt{int}.A$: 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)

- $\overline{A}$ the dual of a session type variable

Duality

$$
\begin{aligned}
\overline{\mathbf{end}} &= \mathbf{end} \\
\overline{(?t.T)} &= !t.\overline{T} \\
\overline{(!t.T)} &= ?t.\overline{T} \\
\overline{\&[\mathtt{l}_i : T_i]_{i\in I}} &= \oplus[\mathtt{l}_i : \overline{T_i}]_{i\in I} \\
\overline{\oplus[\mathtt{l}_i : T_i]_{i\in I}} &= \&[\mathtt{l}_i : \overline{T_i}]_{i\in I} \\
\overline{\overline{A}} &= A
\end{aligned}
$$

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 × Ā
```

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

$$\begin{array}{lll} \texttt{echo\_client} & : & !\alpha.?\beta.\mathtt{end} \to \alpha \to \beta \\ \texttt{echo\_service} & : & ?\alpha.!\alpha.\mathtt{end} \to \mathtt{unit} \end{array}$$

Note that:

$$\overline{!\alpha.?\beta.\mathtt{end}} = !\alpha.?\beta.\mathtt{end} \neq !\alpha.?\alpha.\mathtt{end}$$

However

- $!\alpha.?\beta.\mathtt{end}$ is more general than $!\alpha.?\alpha.\mathtt{end}$
  - Recall that $!\alpha.?\beta.\mathtt{end}$ stands for $\forall\alpha.\forall\beta.!\alpha.?\beta.\mathtt{end}$
- $\forall\alpha.!\alpha.?\alpha.\mathtt{end}$ is a particular instance
- there is a unification for $!\alpha.?\beta.\mathtt{end}$ and $!\alpha.?\alpha.\mathtt{end}$

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

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

- $[\texttt{l}_i : t_i]_{i \in I}$: 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

$$
\begin{array}{lll}
\texttt{val send} & : & \alpha \rightarrow\ !\alpha.A \rightarrow A \\
\texttt{val receive} & : & ?\alpha.A \rightarrow \alpha \times A \\
\texttt{val create} & : & \texttt{unit} \rightarrow A \times \overline{A} \\
\texttt{val close} & : & \texttt{end} \rightarrow \texttt{unit} \\
\texttt{val branch} & : & \&[\texttt{l}_i : A_i]_{i \in I} \rightarrow [\texttt{l}_i : A_i]_{i \in I}
\end{array}
$$

Branch

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

```
val branch : &[l_i : A_i]_{i∈I} → [l_i : A_i]_{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 × Ā
val close   : end → unit
val branch  : &[l_i : A_i]_{i∈I} → [l_i : A_i]_{i∈I}
val select  : (Ā_k → [l_i : Ā_i]_{i∈I}) → ⊕[l_i : A_i]_{i∈I} → A_k
```

$$
\begin{aligned}
&\texttt{val send} &&: \alpha \to\ !\alpha.A \to A\\
&\texttt{val receive} &&: ?\alpha.A \to \alpha \times A\\
&\texttt{val create} &&: \texttt{unit} \to A \times \overline{A}\\
&\texttt{val close} &&: \texttt{end} \to \texttt{unit}\\
&\texttt{val branch} &&: \&[\texttt{l}_i : A_i]_{i\in I} \to [\texttt{l}_i : A_i]_{i\in I}\\
&\texttt{val select} &&: (\overline{A_k} \to [\texttt{l}_i : \overline{A_i}]_{i\in I}) \to \oplus[\texttt{l}_i : A_i]_{i\in I} \to A_k
\end{aligned}
$$

Select

```
val select  : (Ā_k → [l_i : Ā_i]_{i∈I}) → ⊕[l_i : A_i]_{i∈I} → A_k
```

```
opt_echo_client : ⊕[End : end, Msg : !α.?α.end] → bool → α → α
```

```
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
```

$$\texttt{val end\_echo\_client:}\ \oplus[\mathrm{End} : \mathbf{end}] \rightarrow \texttt{unit}$$

$$\texttt{val opt\_echo\_service :}\ \&[\mathrm{End} : \mathbf{end}, \mathrm{Msg} : ?\alpha.!\alpha.\mathbf{end}] \rightarrow \texttt{unit}$$

Note that:

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

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:  ⊕[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.⊕[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
- Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. PPDP'12.

Two types

- $\mathbb{0}$, 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 $\mathbb{0}$ to indicate that no message is respectively received and/or sent

Representation of session types

Encoding

$$
\begin{aligned}
[\![\mathsf{end}]\!] &= \langle \mathbb{0}, \mathbb{0} \rangle \\
[\![?t\,.\,T]\!] &= \langle [\![t]\!] \times [\![T]\!], \mathbb{0} \rangle \\
[\![!t\,.\,T]\!] &= \langle \mathbb{0}, [\![t]\!] \times [\![\overline{T}]\!] \rangle \\
[\![\&[\mathtt{l}_i : T_i]_{i\in I}]\!] &= \langle [\mathtt{l}_i : [\![T_i]\!]]_{i\in I}, \mathbb{0} \rangle \\
[\![\oplus[\mathtt{l}_i : T_i]_{i\in I}]\!] &= \langle \mathbb{0}, [\mathtt{l}_i : [\![\overline{T_i}]\!]]_{i\in I} \rangle \\
[\![A]\!] &= \langle \rho_A, \sigma_A \rangle \\
[\![\overline{A}]\!] &= \langle \sigma_A, \rho_A \rangle
\end{aligned}
$$

Examples

$?\alpha.A$

$$[\![?\alpha.A]\!] = \langle \alpha \times \langle \rho_A, \sigma_A \rangle, \mathbb{0} \rangle$$

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

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

$\overline{T} = \&[\mathrm{End} : \mathrm{end}, \mathtt{Msg} : ?\alpha.!\beta.\mathrm{end}]$

$$\begin{aligned}
[\![\overline{T}]\!] &= \langle [\mathrm{End} : [\![\mathrm{end}]\!], \mathtt{Msg} : [\![?\alpha.!\beta.\mathrm{end}]\!]], \mathbb{0} \rangle \\
&= \langle [\mathrm{End} : \langle \mathbb{0}, \mathbb{0} \rangle, \mathtt{Msg} : \langle \alpha \times [\![!\beta.\mathrm{end}]\!], \mathbb{0} \rangle], \mathbb{0} \rangle \\
&= \langle [\mathrm{End} : \langle \mathbb{0}, \mathbb{0} \rangle, \mathtt{Msg} : \langle \alpha \times \langle \mathbb{0}, \beta \times [\![\mathrm{end}]\!] \rangle, \mathbb{0} \rangle], \mathbb{0} \rangle \\
&= \langle [\mathrm{End} : \langle \mathbb{0}, \mathbb{0} \rangle, \mathtt{Msg} : \langle \alpha \times \langle \mathbb{0}, \beta \times \langle \mathbb{0}, \mathbb{0} \rangle \rangle, \mathbb{0} \rangle], \mathbb{0} \rangle
\end{aligned}$$

Representation of session types

Theorem

If $[\![T]\!] = \langle t, s \rangle$, then $[\![\overline{T}]\!] = \langle s, t \rangle$.

Interface in Ocaml

Session

```
module Session : sig
  type 𝟘
  type (ρ,σ) st (* OCaml syntax for ⟨ρ,σ⟩ *)
  val create  : unit → (ρ,σ) st × (σ,ρ) st
  val close   : (𝟘,𝟘) st → unit
  val send    : α → (𝟘,(α × (σ,ρ) st)) st → (ρ,σ) st
  val receive : ((α × (ρ,σ) st),𝟘) st → α × (ρ,σ) st
  val select  : ((σ,ρ) st → α) → (𝟘,[>] as α) st → (ρ,σ) st
  val branch  : ([>] as α,𝟘) 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 (α,β) 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   : (𝟘,𝟘) 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 → (𝟘,(α × (σ,ρ) st)) st → (ρ,σ) st
val receive : ((α × (ρ,σ) st),𝟘) 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) → (𝟘,[>] as α) st → (ρ,σ) st
val branch  : ([>] as α,𝟘) 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 :
  (𝟘,[> `End of (𝟘,𝟘) st
     |  `Msg of (β × (𝟘,γ × (𝟘,α) st) st,𝟘) st]
     as α) st → β list → γ list
```

The session type

```
val rec_echo_client :
  rec X.⊕[ End: end | Msg: !α.?β.X ] →
  α list → β list
```

is obtained by encoding back the representation[2]

[2] pretty 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 : !α.?α. → α → α → β
val service : ?α.!β. → unit
```

Its execution raises the exception `Session.InvalidEndpoint`