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

[1] Padovani, 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

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

- `FuSe` provides polymorphic session types
- α is a type variable
- A is a session type variable
- $\overline{A}$ the dual of a session type variable

Session Types

Syntax

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

- $[\texttt{l}_i : t_i]_{i \in I}$: Variants (disjoint sums)

Duality

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

An API for sessions

Module Session

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

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

$?\alpha.A$

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

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

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

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

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

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
```

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`