Binary Sessions + DbC

Hernán Melgratti

ICC University of Buenos Aires-Conicet

Binary Sessions + DbC

- An extension of `FuSe` with dynamically checked contracts that states properties[1]
  - about exchanged messages
  - the structure of the protocol

[1] M., Luca Padovani: Chaperone contracts for higher-order sessions. PACMPL 1(ICFP).

FuSe + Service channels (shared channels)

```
module type Service = sig
  type α t
  val register : ((β, α) st → unit) → (α, β) st t
  val connect  : (α, β) st t → (α, β) st
end
```

- α is the session type from the client's viewpoint
- `register f` creates a new shared channel and registers the service `f` to it.
  - Each connection spawns a new thread running `f`
  - returns the shared channel
- `connect ch` connects with the service on the shared channel `ch`
  - return the client endpoint of the established session.

FuSe + Service channels (shared channels)

Roots of a polynomial

```
let server ep =
  let p, ep = receive ep in
  let root = ... in
  let ep = send root ep in
  close ep

let math_service = register server
```

```
val server : ?poly.!float.end → unit
val math_service : !poly.?float.end Service.t
```

```
let user () =
  let ep = connect math_service in
  let ep = send (from_list [2.0; -3.0; 1.0]) ep in
  let _, ep = receive ep in
  close ep
```

A simple FuSe program + Contracts

Roots of a polynomial

```
let server ep =
  let p, ep = receive ep in
  let root = ... in (* assumes p is a linear equation *)
  let ep = send root ep in
  close ep

let math_service = register server contract "Server"
                   (*service with a contract and a blame label *)

let user () =
  let ep = connect math_service "Client" in
  let ep = send (from_list [2.0; -3.0; 1.0]) ep in
  let _, ep = receive ep in
  close ep
```

Language for Contracts

Constructors

$$\begin{aligned}
\texttt{flat_c} &: (t \to \texttt{bool}) \to \mathsf{con}(t) && t :: \omega \\
\\
\texttt{send_c} &: \mathsf{con}(t) \to \mathsf{con}(T) \to \mathsf{con}(!t.\,T) \\
\texttt{receive_c} &: \mathsf{con}(t) \to \mathsf{con}(T) \to \mathsf{con}(?t.\,T) \\
\\
\texttt{end_c} &: \mathsf{con}(\mathsf{end})
\end{aligned}$$

Dependent Contracts

Roots of a polynomial

```
let degree p = ... (* computes the degree of a polynomial *)

let contract = send_c  (flat_c (fun p → degree p == 1))  @@
                  ...    (* contract for the continuation *)
```

Contracts

Roots of a polynomial

```
let contract = send_c  (flat_c (fun p → degree p == 1))  @@
               receive_c (flat_c (fun _ → true)) @@
               end_c
```

- The continuation does not impose any restriction to the communication protocol
- … but tedious to write

any_c

Constructors

$$\begin{aligned}
\texttt{flat_c} &: (t \to \texttt{bool}) \to \mathsf{con}(t) && t :: \omega \\
\texttt{send_c} &: \mathsf{con}(t) \to \mathsf{con}(T) \to \mathsf{con}(!t.T) \\
\texttt{receive_c} &: \mathsf{con}(t) \to \mathsf{con}(T) \to \mathsf{con}(?t.T) \\
\texttt{end_c} &: \mathsf{con}(\texttt{end}) \\
\texttt{any_c} &: \mathsf{con}(\alpha)
\end{aligned}$$

Roots of a polynomial

```
let contract = send_c  (flat_c (fun p → degree p == 1))  @@
               any_c  (* trivial contract  *)
```

- Can we give some guarantee about the response?
- We would like to specify that the response is a root of the polynomial

Dependent Contracts

Constructors

$$
\begin{aligned}
\texttt{flat\_c} &: (t \to \texttt{bool}) \to \mathsf{con}(t) && t :: \omega \\[1ex]
\texttt{send\_c} &: \mathsf{con}(t) \to \mathsf{con}(T) \to \mathsf{con}(!t\,.\,T) \\
\texttt{receive\_c} &: \mathsf{con}(t) \to \mathsf{con}(T) \to \mathsf{con}(?t\,.\,T) \\[1ex]
\texttt{end\_c} &: \mathsf{con}(\texttt{end}) \\[1ex]
\texttt{any\_c} &: \mathsf{con}(\alpha) \\[1ex]
\texttt{send\_d} &: \mathsf{con}(t) \to (t \to \mathsf{con}(T)) \to \mathsf{con}(!t\,.\,T) && t :: \omega \\
\texttt{receive\_d} &: \mathsf{con}(t) \to (t \to \mathsf{con}(T)) \to \mathsf{con}(?t\,.\,T) && t :: \omega
\end{aligned}
$$

Contracts

Roots of a polynomial

```
let root_of p r = ... (* check if r is a root of p *)

let contract = send_d (flat_c (fun p → degree p == 1)) @@
               fun p → receive_c (flat_c (root_of p)) @@
               end_c
```

Contracts for choices

Simplified version of choices

$$\begin{aligned} \texttt{left} &: T \oplus S \to T \\ \texttt{right} &: T \oplus S \to S \\ \texttt{branch} &: T \mathbin{\&} S \to T + S \end{aligned}$$

```
type α + β = [  Left of α |  Right of β ]
val left : (𝟘, (ρ1, σ1) st + (ρ2, σ2) st)  → (σ1, ρ1) st
val right : (𝟘, (ρ1, σ1) st + (ρ2, σ2) st)  → (σ2, ρ2) st
val branch : ((ρ1, σ1) st + (ρ2, σ2) st,𝟘)
                                           → (ρ1, σ1) st + (ρ2, σ2) st
```

```
let left ep = send true ep
let right ep = send false ep
let branch ep =
  use ep;
  if UnsafeChannel.receive ep.channel
  then  Left (fresh ep)
  else  Right (fresh ep)
```

Contracts for choices

Constructors

$$
\begin{array}{rcll}
\texttt{flat\_c} & : & (t \to \texttt{bool}) \to \mathsf{con}(t) & t :: \omega \\
\\
\texttt{send\_c} & : & \mathsf{con}(t) \to \mathsf{con}(T) \to \mathsf{con}(\texttt{!}t\texttt{.}T) & \\
\texttt{receive\_c} & : & \mathsf{con}(t) \to \mathsf{con}(T) \to \mathsf{con}(\texttt{?}t\texttt{.}T) & \\
\\
\texttt{end\_c} & : & \mathsf{con}(\mathsf{end}) & \\
\\
\texttt{any\_c} & : & \mathsf{con}(\alpha) & \\
\\
\texttt{send\_d} & : & \mathsf{con}(t) \to (t \to \mathsf{con}(T)) \to \mathsf{con}(\texttt{!}t\texttt{.}T) & t :: \omega \\
\texttt{receive\_d} & : & \mathsf{con}(t) \to (t \to \mathsf{con}(T)) \to \mathsf{con}(\texttt{?}t\texttt{.}T) & t :: \omega \\
\\
\texttt{choice\_c} & : & \mathsf{con}(\texttt{bool}) \to \mathsf{con}(T) \to \mathsf{con}(S) \to \mathsf{con}(T \oplus S) & \\
\texttt{branch\_c} & : & \mathsf{con}(\texttt{bool}) \to \mathsf{con}(T) \to \mathsf{con}(S) \to \mathsf{con}(T \& S) &
\end{array}
$$

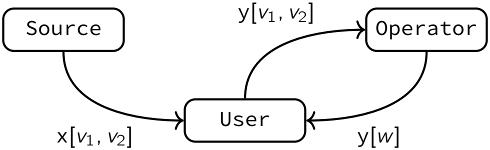
Contracts for choices

Roots of a polynomial

```
let server ep =
  let p, ep = receive ep in
  (* it sends as many messages as the real roots of p *)
  ...
val server :  ?poly.rec A.(!float.A ⊕ end)-> unit

let contract =
  send_d (flat_c (fun p → degree p > 0)) @@
  fun p →
      let rec missing_roots n =
        if n > 0 then
          branch_c
            any_c
            (receive_c (flat_c (root_of p)) @@
               missing_roots (n - 1))
            end_c
        else
          branch_c (flat_c not) any_c end_c
      in missing_roots (degree p)
```

First order interaction and blame




```
x : ?int.?int.end          y : !int.!int.?int.end
                           op_c = send_c any_c @@
src_c =  any_c                    send_c (flat_c ((<>) 0)) @@
                                  receive_c (flat_c (>= 0)) @@ end_c
```

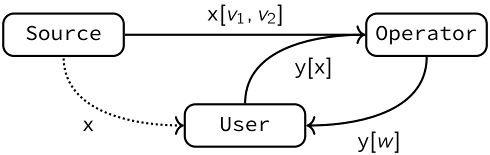
First order interaction and blame

First order user

```
let user () =
  let x = connect source_chan "User" in
  let y = connect operator_chan "User" in
  let v1, x = receive x in
  let v2, x = receive x in
  let y = send v1 y in
  let y = send v2 y in
  let w, y = receive y in
  print_int w; close x; close y
```

Which party should be blamed if $v2 < 0$? User

Higher-order communication and blame



x : ?int.?int.end

```
src_c =  any_c
```

y : !(?int.?int.end).?int.end

```
op_c = send_c d_c @@
       receive_c (flat_c ((<=) 0)) @@
       end_c
d_c =  receive_c any_c @@
       receive_c (flat_c ((<>) 0)) @@
       end_c
```

Higher-order communication and blame

Delegating user

```
let user_deleg () =
  let x = connect source_chan "User" in
  let y = connect operator_deleg_chan "User" in
  let y = send x y in
  let res, y = receive y in
  print_int res; close y
```

Which party should be blamed if te second value generated by `source_chan` is negative? User (despite it is not involved in the communication)

λCoS

Syntax

Expression	e	$::=$	v	value
		$\mid$	x	variable
		$\mid$	$e_1\, e_2$	application
		$\mid$	$\mathtt{let}\ x, y = e_1\ \mathtt{in}\ e_2$	pair splitting
		$\mid$	$\mathtt{case}\ e\ \mathtt{of}\ e_1 \mid e_2$	case analysis
		$\mid$	$\mathtt{mon}^{k,l}(e_2, e_1)$	monitor
		$\mid$	$v \triangleleft^{k} e$	busy monitor
		$\mid$	$\mathtt{blame}\ k$	blame
Value	v, w, κ_1, κ_2	$::=$	$\mathtt{c}^{n}\ v_1 \cdots v_n$	applied constant
		$\mid$	$\lambda x.e$	abstraction
		$\mid$	ε	endpoint
Process	P, Q	$::=$	$\langle e \rangle_k$	thread
		$\mid$	$P \parallel Q$	composition
		$\mid$	$a \Leftarrow^{\kappa_1}_{k} v$	service
		$\mid$	$(\nu a)P$	session
Endpoint	ε	$::=$	a^{p}	lone endpoint
		$\mid$	$\mathtt{mon}^{k,l}(\kappa_1, \varepsilon)$	monitored endpoint

λCoS

Constants

c^n	n max	Sugared	Description
()	0		unit
true, false	0		boolean values
pair	2	(v,w)	pair creation
inl, inr	1		left/right injection
fix	0		fixpoint combinator
connect	0		initiate session
close	0		terminate session
receive	0		input
send	1		output
branch	0		offer choice
left	0		choose left
right	0		choose right
flat_c	1		flat contract
end_c	0		closed endpoint
receive_c	2	$?\kappa_1.\kappa_2$	non-dependent input
send_c	2	$!\kappa_1.\kappa_2$	non-dependent output
receive_d	2	$?\kappa_1 \mapsto w$	dependent input
send_d	2	$!\kappa_1 \mapsto w$	dependent output
branch_c	3	$?\kappa_1 \mapsto \kappa_2 : \kappa_3$	external choice
choice_c	3	$!\kappa_1 \mapsto \kappa_2 : \kappa_3$	internal choice
dual	0		compute dual contract

Typing of λCoS

Types

Session Type $T, S ::= \mathsf{end} \mid\ !t\,.\,T \mid\ ?t\,.\,T \mid T \oplus S \mid T \,\&\, S$

Type $t, s ::= \mathtt{unit} \mid \mathtt{bool} \mid t \rightarrow^{\iota} s \mid t + s \mid T \mid \mathsf{con}(t) \mid t \times s \mid \# T$

Kind $\iota ::= 1 \mid \omega$

λCoS

Type schemes of λCoS constants

$$
\begin{array}{rcll}
() & : & \mathtt{unit} & \\
\mathtt{true}, \mathtt{false} & : & \mathtt{bool} & \\
\mathtt{pair} & : & t \to s \to^{\iota} t \times s & t :: \iota \\
\mathtt{inl} & : & t \to t + s & \\
\mathtt{inr} & : & s \to t + s & \\
\mathtt{close} & : & \mathtt{end} \to \mathtt{unit} & \\
\mathtt{send} & : & t \to \mathtt{!}t.T \to^{\iota} T & t :: \iota \\
\mathtt{receive} & : & \mathtt{?}t.T \to t \times T & \\
\mathtt{left} & : & T \oplus S \to T & \\
\mathtt{right} & : & T \oplus S \to S & \\
\mathtt{branch} & : & T \mathbin{\&} S \to T + S & \\
\mathtt{connect} & : & \#T \to T & \\
\mathtt{flat\_c} & : & (t \to \mathtt{bool}) \to \mathrm{con}(t) & t :: \omega \\
\mathtt{end\_c} & : & \mathrm{con}(\mathtt{end}) & \\
\mathtt{send\_c} & : & \mathrm{con}(t) \to \mathrm{con}(T) \to \mathrm{con}(\mathtt{!}t.T) & \\
\mathtt{receive\_c} & : & \mathrm{con}(t) \to \mathrm{con}(T) \to \mathrm{con}(\mathtt{?}t.T) & \\
\mathtt{send\_d} & : & \mathrm{con}(t) \to (t \to \mathrm{con}(T)) \to \mathrm{con}(\mathtt{!}t.T) & t :: \omega \\
\mathtt{receive\_d} & : & \mathrm{con}(t) \to (t \to \mathrm{con}(T)) \to \mathrm{con}(\mathtt{?}t.T) & t :: \omega \\
\mathtt{choice\_c} & : & \mathrm{con}(\mathtt{bool}) \to \mathrm{con}(T) \to \mathrm{con}(S) \to \mathrm{con}(T \oplus S) & \\
\mathtt{branch\_c} & : & \mathrm{con}(\mathtt{bool}) \to \mathrm{con}(T) \to \mathrm{con}(S) \to \mathrm{con}(T \mathbin{\&} S) & \\
\mathtt{dual} & : & \mathrm{con}(T) \to \mathrm{con}(\overline{T}) &
\end{array}
$$

Typing

Typing rules for expressions $\boxed{\Gamma \vdash e : t}$

[t-const]
$$\frac{t \in \mathrm{typeof}(\mathtt{c}) \qquad \Gamma :: \omega}{\Gamma \vdash \mathtt{c} : t}$$

[t-name]
$$\frac{\Gamma :: \omega}{\Gamma, u : t \vdash u : t}$$

[t-fun]
$$\frac{\Gamma, x : t \vdash e : s \qquad \Gamma :: \iota}{\Gamma \vdash \lambda x \,.\, e : t \to^{\iota} s}$$

[t-app]
$$\frac{\Gamma_1 \vdash e_1 : t \to^{\iota} s \qquad \Gamma_2 \vdash e_2 : t}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : s}$$

[t-split]
$$\frac{\Gamma_1 \vdash e_1 : t_1 \times t_2 \qquad \Gamma_2, x : t_1, y : t_2 \vdash e_2 : t}{\Gamma_1 + \Gamma_2 \vdash \mathtt{let}\ x, y = e_1\ \mathtt{in}\ e_2 : t}$$

[t-case]
$$\frac{\Gamma_1 \vdash e : t_1 + t_2 \qquad \Gamma_2 \vdash e_i : t_i \to^{\iota_i} t \ ^{(i=1,2)}}{\Gamma_1 + \Gamma_2 \vdash \mathtt{case}\ e\ \mathtt{of}\ e_1 \mid e_2 : t}$$

[t-blame]
$$\Gamma \vdash \mathtt{blame}\ k : t$$

[t-monitor]
$$\frac{\Gamma_1 \vdash e_1 : t \qquad \Gamma_2 \vdash e_2 : \mathrm{con}(t)}{\Gamma_1 + \Gamma_2 \vdash \mathrm{mon}^{k,l}(e_2, e_1) : t}$$

[t-busy-monitor]
$$\frac{\Gamma_1 \vdash e : \mathtt{bool} \qquad \Gamma_2 \vdash v : t}{\Gamma_1 + \Gamma_2 \vdash v \lhd^{k} e : t}$$

Typing

Typing rules for processes

$\boxed{\Gamma \vdash P}$

[t-thread]

$$\frac{\Gamma \vdash e : \texttt{unit}}{\Gamma \vdash \langle e \rangle_k}$$

[t-par]

$$\frac{\Gamma_i \vdash P_i \ ^{(i=1,2)}}{\Gamma_1 + \Gamma_2 \vdash P_1 \parallel P_2}$$

[t-session]

$$\frac{\Gamma, a^{+} : T, a^{-} : \overline{T} \vdash P}{\Gamma \vdash (\nu a)P}$$

[t-service]

$$\frac{\emptyset \vdash \kappa_1 : \mathsf{con}(T) \quad \Gamma \vdash v : \overline{T} \to \texttt{unit}}{\Gamma + a : \# T \vdash a \Leftarrow_k^{\kappa_1} v}$$

Reduction of expressions (1)

$$
\begin{array}{lrcll}
[\mathtt{r-beta}] & (\lambda x.e)\,v & \rightarrow & e\{v/x\} & \\
[\mathtt{r-split}] & \mathtt{let}\; x,y = (v,w)\; \mathtt{in}\; e & \rightarrow & e\{v,w/x,y\} & \\
[\mathtt{r-inl}] & \mathtt{case}\; \mathtt{inl}\; v\; \mathtt{of}\; e_1 \mid e_2 & \rightarrow & e_1\, v & \\
[\mathtt{r-inr}] & \mathtt{case}\; \mathtt{inr}\; v\; \mathtt{of}\; e_1 \mid e_2 & \rightarrow & e_2\, v & \\
[\mathtt{r-flat}] & \mathtt{mon}^{k,l}(\mathtt{flat\_c}\; w, v) & \rightarrow & v \lhd^k wv & \\
[\mathtt{r-true}] & v \lhd^k \mathtt{true} & \rightarrow & v & \\
[\mathtt{r-false}] & v \lhd^k \mathtt{false} & \rightarrow & \mathtt{blame}\; k & \\
[\mathtt{r-context}] & \mathscr{E}[e] & \rightarrow & \mathscr{E}[e'] & \text{if } e \rightarrow e'
\end{array}
$$

$$\mathscr{E} ::= [\,] \mid \mathscr{E} e \mid v\mathscr{E} \mid \mathtt{mon}^{\sigma}(e, \mathscr{E}) \mid v \lhd^k \mathscr{E} \mid \mathtt{let}\; x,y = \mathscr{E}\; \mathtt{in}\; e \mid \mathtt{case}\; \mathscr{E}\; \mathtt{of}\; e_1 \mid e_2 \mid \mathtt{mon}^{\sigma}(\mathscr{E}, v)$$

Semantics

Session establishment

[r − connect]

$$\begin{pmatrix} \langle \mathscr{E}[\texttt{connect}\ a] \rangle_k \\ a \Leftarrow_l^{\kappa_1} v \end{pmatrix} \rightarrow (\nu b) \begin{pmatrix} \langle \mathscr{E}[\mathsf{mon}^{l,k}(\kappa_1, b^+)] \rangle_k \\ \langle v\ \mathsf{mon}^{k,l}(\texttt{dual}\ \kappa_1, b^-) \rangle_l \end{pmatrix} \parallel a \Leftarrow_l^{\kappa_1} v \quad b \text{ fresh}$$

$$\mathscr{E} ::= [\,] \mid \mathscr{E} e \mid v\mathscr{E} \mid \mathsf{mon}^{\sigma}(e, \mathscr{E}) \mid v \triangleleft^k \mathscr{E} \mid \texttt{let}\ x\texttt{,}y = \mathscr{E}\ \texttt{in}\ e \mid \texttt{case}\ \mathscr{E}\ \texttt{of}\ e_1 \mid e_2 \mid \mathsf{mon}^{\sigma}(\mathscr{E}, v)$$

Reduction of expressions (2)

[d – end]	$\mathtt{dual}\ \mathtt{end\_c}$	$\rightarrow$	$\mathtt{end\_c}$
[d – send – c]	$\mathtt{dual}\ !\kappa_1.\kappa_2$	$\rightarrow$	$?\kappa_1.(\mathtt{dual}\ \kappa_2)$
[d – receive – c]	$\mathtt{dual}\ ?\kappa_1.\kappa_2$	$\rightarrow$	$!\kappa_1.(\mathtt{dual}\ \kappa_2)$
[d – send – d]	$\mathtt{dual}\ !\kappa_1 \mapsto w$	$\rightarrow$	$?\kappa_1 \mapsto (\lambda x.\mathtt{dual}\ (wx))$
[d – receive – d]	$\mathtt{dual}\ ?\kappa_1 \mapsto w$	$\rightarrow$	$!\kappa_1 \mapsto (\lambda x.\mathtt{dual}\ (wx))$
[d – choice]	$\mathtt{dual}\ !\kappa_1 \mapsto \kappa_2 : \kappa_3$	$\rightarrow$	$?\kappa_1 \mapsto (\mathtt{dual}\ \kappa_2) : (\mathtt{dual}\ \kappa_3)$
[d – branch]	$\mathtt{dual}\ ?\kappa_1 \mapsto \kappa_2 : \kappa_3$	$\rightarrow$	$!\kappa_1 \mapsto (\mathtt{dual}\ \kappa_2) : (\mathtt{dual}\ \kappa_3)$

$$\mathscr{E} ::= [\,] \mid \mathscr{E}e \mid v\mathscr{E} \mid \mathrm{mon}^\sigma(e, \mathscr{E}) \mid v \lhd^k \mathscr{E} \mid \mathtt{let}\ x, y = \mathscr{E}\ \mathtt{in}\ e \mid \mathtt{case}\ \mathscr{E}\ \mathtt{of}\ e_1 \mid e_2 \mid \mathrm{mon}^\sigma(\mathscr{E}, v)$$

Semantics

Communication (simplified)

$$[\mathrm{r-comm}] \quad \begin{pmatrix} \langle \mathscr{E}[\texttt{send}\ v\ \texttt{mon}^{\sigma}(!\kappa_1 . \kappa_2, a^{p})]\rangle_k \\ \langle \mathscr{E}'[\texttt{receive}\ \texttt{mon}^{\varrho}(?\kappa_3 . \kappa_4, a^{\overline{p}})]\rangle_l \end{pmatrix} \quad \rightarrow$$

$$\begin{pmatrix} \langle \mathscr{E}[\texttt{mon}^{\sigma}(\kappa_2, a^{p})]\rangle_k \\ \langle \mathscr{E}'[(\texttt{mon}^{\varrho}(\kappa_3, \texttt{mon}^{\neg\sigma}(\kappa_1, v)), \texttt{mon}^{\varrho}(\kappa_4, a^{\overline{p}}))]\rangle_l \end{pmatrix}$$

where $\neg(k, l) = l, k$

- Note that v can be of a non basic type, hence the monitor cannot be evaluated.
- Endpoints have a stack of monitors

$$\texttt{mon}^{\vec{\sigma}}(\vec{\kappa}, e) \quad \text{for} \quad \texttt{mon}^{\sigma_n}(\kappa_n, \cdots \texttt{mon}^{\sigma_1}(\kappa_1, e)\cdots)$$

$$\texttt{mon}^{\overleftarrow{\sigma}}(\overleftarrow{\kappa}, e) \quad \text{for} \quad \texttt{mon}^{\sigma_1}(\kappa_1, \cdots \texttt{mon}^{\sigma_n}(\kappa_n, e)\cdots)$$

Semantics

Communication

$$[\mathsf{r-comm}]$$

$$\begin{pmatrix} \langle \mathscr{E}[\texttt{send}\ v\ \mathsf{mon}^{\vec{\sigma}}(\overrightarrow{\texttt{!}\kappa_1\texttt{.}\kappa_2}, a^p)]\rangle_k \\ \langle \mathscr{E}'[\texttt{receive}\ \mathsf{mon}^{\vec{\varrho}}(\overrightarrow{\texttt{?}\kappa_3\texttt{.}\kappa_4}, a^{\overline{p}})]\rangle_l \end{pmatrix} \quad \rightarrow$$

$$\begin{pmatrix} \langle \mathscr{E}[\mathsf{mon}^{\vec{\sigma}}(\overrightarrow{\kappa_2}, a^p)]\rangle_k \\ \langle \mathscr{E}'[\texttt{(}\mathsf{mon}^{\vec{\varrho}}(\overrightarrow{\kappa_3}, \mathsf{mon}^{\overleftarrow{\neg\sigma}}(\overleftarrow{\kappa_1}, v))\texttt{,}\mathsf{mon}^{\vec{\varrho}}(\overrightarrow{\kappa_4}, a^{\overline{p}})\texttt{)}]\rangle_l \end{pmatrix}$$

Semantics

Dependent communication

$$[\mathrm{r-comm-d}]$$

$$\begin{pmatrix} \langle \mathscr{E}[\texttt{send}\ v\ \texttt{mon}^{\vec{\vec{\sigma}}}(\overrightarrow{!\kappa_1 \mapsto w_1}, a^{p})]\rangle_k \\ \langle \mathscr{E}'[\texttt{receive}\ \texttt{mon}^{\vec{\vec{\varrho}}}(\overrightarrow{?\kappa_2 \mapsto w_2}, a^{\overline{p}})]\rangle_l \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} \langle \mathscr{E}[\texttt{mon}^{\vec{\vec{\sigma}}}(\overrightarrow{w_1\,v}, a^{p})]\rangle_k \\ \langle \mathscr{E}'[\texttt{(mon}^{\vec{\vec{\varrho}}}(\overrightarrow{\kappa_2}, \texttt{mon}^{\overleftarrow{\neg\sigma}}(\overleftarrow{\kappa_1}, v)\texttt{)}\texttt{,}\texttt{mon}^{\vec{\vec{\varrho}}}(\overrightarrow{w_2\,v}, a^{\overline{p}})\texttt{)}]\rangle_l \end{pmatrix}$$

Semantics

Choices

$$[\mathtt{r-left}]$$

$$\begin{pmatrix} \langle \mathscr{E}[\mathtt{left}\ \mathtt{mon}^{\vec{\sigma}}(\overrightarrow{!\kappa_1 \mapsto \kappa_2 : \kappa_3}, a^p)] \rangle_k \\ \langle \mathscr{E}'[\mathtt{branch}\ \mathtt{mon}^{\vec{\varrho}}(\overrightarrow{?\kappa_4 \mapsto \kappa_5 : \kappa_6}, a^{\overline{p}})] \rangle_l \end{pmatrix} \quad \to$$

$$\begin{pmatrix} \langle \mathscr{E}[\mathtt{mon}^{\vec{\sigma}}(\overrightarrow{\kappa_2}, a^p)] \rangle_k \\ \langle \mathscr{E}'[(\lambda\_.\,\mathtt{inl}\ \mathtt{mon}^{\vec{\varrho}}(\overrightarrow{\kappa_5}, a^{\overline{p}}))\ \mathtt{mon}^{\vec{\varrho}}(\overrightarrow{\kappa_4}, \mathtt{mon}^{\neg\sigma}(\overleftarrow{\kappa_1}, \mathtt{true}))] \rangle_l \end{pmatrix}$$

$$[\mathtt{r-right}]$$

$$\begin{pmatrix} \langle \mathscr{E}[\mathtt{right}\ \mathtt{mon}^{\vec{\sigma}}(\overrightarrow{!\kappa_1 \mapsto \kappa_2 : \kappa_3}, a^p)] \rangle_k \\ \langle \mathscr{E}'[\mathtt{branch}\ \mathtt{mon}^{\vec{\varrho}}(\overrightarrow{?\kappa_4 \mapsto \kappa_5 : \kappa_6}, a^{\overline{p}})] \rangle_l \end{pmatrix} \quad \to$$

$$\begin{pmatrix} \langle \mathscr{E}[\mathtt{mon}^{\vec{\sigma}}(\overrightarrow{\kappa_2}, a^p)] \rangle_k \\ \langle \mathscr{E}'[(\lambda\_.\,\mathtt{inr}\ \mathtt{mon}^{\vec{\varrho}}(\overrightarrow{\kappa_5}, a^{\overline{p}}))\ \mathtt{mon}^{\vec{\varrho}}(\overrightarrow{\kappa_4}, \mathtt{mon}^{\neg\sigma}(\overleftarrow{\kappa_1}, \mathtt{false}))] \rangle_l \end{pmatrix}$$

Semantics

Session termination

$$[\mathtt{r-close}]$$

$$(\nu a)\begin{pmatrix}\langle \mathscr{E}[\mathtt{close}\ \mathsf{mon}^{\vec{\sigma}}(\overrightarrow{\mathtt{end\_c}}, a^{+})]\rangle_k \\ \langle \mathscr{E}'[\mathtt{close}\ \mathsf{mon}^{\vec{\varrho}}(\overrightarrow{\mathtt{end\_c}}, a^{-})]\rangle_l\end{pmatrix} \rightarrow \langle \mathscr{E}[()]\rangle_k \parallel \langle \mathscr{E}'[()]\rangle_l$$

Properties

Subject reduction

- If $\Gamma :: \omega$ and $\Gamma \vdash P$ and $P \to Q$, then $\Gamma \vdash Q$.
- If Γ is balanced and $\Gamma \vdash P$ and $P \to Q$, then there exists Γ' such that $\Gamma \to^* \Gamma'$ and $\Gamma' \vdash Q$.

Blame safety

Goal

- to ensure that a process that *honours it contracts* cannot be blamed
  - Roughly, if a process sends a value, it is one accepted by the contracts of the monitored channel.
  - ... the formal definition is involved because of dependent contracts and delegation

Contract entailment

$\kappa_1 \leqslant \kappa_2$ if each value that satisfies κ_1 also satisfies κ_2

$$\texttt{flat\_c}\ (\geq 3) \leqslant \texttt{flat\_c}\ (\geq 0)$$

Contract entailment

$e_1 \leqslant e_2$ implies either:

1. $e_1 \Downarrow \texttt{flat\_c}\ w_1$ and $e_2 \Downarrow \texttt{flat\_c}\ w_2$ and for every $v \in w_1$ we have $v \in w_2$, or
2. $e_1 \Downarrow \texttt{end\_c}$ and $e_2 \Downarrow \texttt{end\_c}$, or
3. $e_1 \Downarrow \texttt{!}\kappa_1\texttt{.}\kappa_2$ and $e_2 \Downarrow \texttt{!}\kappa_3\texttt{.}\kappa_4$ and $\kappa_3 \leqslant \kappa_1$ and $\kappa_2 \leqslant \kappa_4$, or
4. $e_1 \Downarrow \texttt{?}\kappa_1\texttt{.}\kappa_2$ and $e_2 \Downarrow \texttt{?}\kappa_3\texttt{.}\kappa_4$ and $\kappa_1 \leqslant \kappa_3$ and $\kappa_2 \leqslant \kappa_4$, or
5. ...

Locally correctness

$k \mathscr{C} P$: k is (locally) correct in P

1. $P = \mathscr{P}_k[\mathtt{send}\ v\ \mathtt{mon}^{-,-}(!\mathtt{flat\_c}\ w\,\mathtt{.}\,\_, \_)]$ implies $v \in w$, and
2. $P = \mathscr{P}_k[\mathtt{send}\ v\ \mathtt{mon}^{-,-}(!\mathtt{flat\_c}\ w \mapsto \_, \_)]$ implies $v \in w$, and
3. $P = \mathscr{P}_k[\mathtt{send}\ \mathtt{mon}^{-,-}(\kappa_1, \varepsilon)\ \mathtt{mon}^{-,-}(!\kappa_2\,\mathtt{.}\,\_, \_)]$ implies $\kappa_1 \leqslant \kappa_2$, and
4. $P = \mathscr{P}_k[\mathtt{left}\ \mathtt{mon}^{-,-}(!\mathtt{flat\_c}\ w \mapsto \_\mathtt{:}\_, \_)]$ implies $\mathtt{true} \in w$, and
5. $P = \mathscr{P}_k[\mathtt{right}\ \mathtt{mon}^{-,-}(!\mathtt{flat\_c}\ w \mapsto \_\mathtt{:}\_, \_)]$ implies $\mathtt{false} \in w$, and
6. $P \rightarrow Q$ implies $k \mathscr{C} Q$.

$$\mathscr{P}_k \quad ::= \quad \langle \mathscr{E} \rangle_k \quad | \quad (\mathscr{P}_k \parallel P) \quad | \quad (P \parallel \mathscr{P}_k) \quad | \quad (\nu a)\mathscr{P}_k$$

Property

Blame safety

If $\Gamma \vdash P$ where P is a user process and k is locally correct in P, then $P \rightarrow^* Q$ implies $\texttt{blame}\ k \not\subset Q$.

Implementation of contracts

GADT for contracts

type [_] =
  | Flat : $(\alpha \to \texttt{bool}) \to [\alpha]$
  | End : [end]
  | Receive : $[\alpha] \times (\alpha \to [A]) \to [?\alpha.A]$
  | Send : $[\alpha] \times (\alpha \to [A]) \to [!\alpha.A]$
  | Branch : $[\texttt{bool}] \times [A] \times [B] \to [A \,\&\, B]$
  | Choice : $[\texttt{bool}] \times [A] \times [B] \to [A \oplus B]$

Implementation of contracts

Contract primitives

```
let flat_c w = Flat w
let any_c = Flat (fun _ → true)
let receive_d k f = Receive (k, f)
let receive_c k1 k2 = receive_d k1 (fun _ → k2)
...
```

Implementation of contracts

Monitored endpoint

```
type A mt =
  | Channel of linearity_tag_type × A st
  | Monitor of [⟨α,β⟩] × string × string × ⟨α,β⟩
```

Implementation of contracts

Implementation of primitives

```
let rec send v =
  function
  | Channel (lin, ep) → Channel (lin, FuSe.send v ep)
  | Monitor (Send (k, w), pos, neg, ep) →
     wrap (w v) pos neg (send (wrap k neg pos v) ep)
  | Monitor (Flat _, _, _, _) → assert false (*IMPOSSIBLE*)

let wrap : type a. [a] → string → string → a → a
  = fun k pos neg v →
  match k with
  | Flat w          → if unlimited v && w v
                      then v else raise (Blame pos)
  | End as k        → Monitor (k, pos, neg, v)
  | Receive _ as k → Monitor (k, pos, neg, v)
  | Send _ as k     → Monitor (k, pos, neg, v)
  | Branch _ as k   → Monitor (k, pos, neg, v)
  | Choice _ as k   → Monitor (k, pos, neg, v)
```