Binary Sessions + DbC

Hernán Melgratti

ICC University of Buenos Aires-Conicet

20 February 2020 @ Pisa

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
```

FuSe + Service channels

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

```
module Service : ServiceSig = struct
  type α t = UnsafeChannel.t

  let register f =
    let ch = UnsafeChannel.create () in
    let rec server () =
      let _ = Thread.create f (UnsafeChannel.receive ch) in
      server ()
    in
    let _ = Thread.create server () in ch

  let connect ch =
    let a, b = FuSe.create () in
    UnsafeChannel.send a ch;
    b
end
```

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

$$
\begin{aligned}
\texttt{flat\_c} &: (t \to \texttt{bool}) \to \mathsf{con}(t) \qquad\qquad t :: \omega \\
\\
\texttt{send\_c} &: \mathsf{con}(t) \to \mathsf{con}(T) \to \mathsf{con}(!t.T) \\
\texttt{receive\_c} &: \mathsf{con}(t) \to \mathsf{con}(T) \to \mathsf{con}(?t.T) \\
\\
\texttt{end\_c} &: \mathsf{con}(\texttt{end})
\end{aligned}
$$

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

$$
\begin{aligned}
\texttt{flat\_c} &: (t \rightarrow \texttt{bool}) \rightarrow \texttt{con}(t) && t :: \omega \\
\texttt{send\_c} &: \texttt{con}(t) \rightarrow \texttt{con}(T) \rightarrow \texttt{con}(!t.T) \\
\texttt{receive\_c} &: \texttt{con}(t) \rightarrow \texttt{con}(T) \rightarrow \texttt{con}(?t.T) \\
\texttt{end\_c} &: \texttt{con}(\texttt{end}) \\
\texttt{any\_c} &: \texttt{con}(\alpha)
\end{aligned}
$$

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

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

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

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

```
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

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

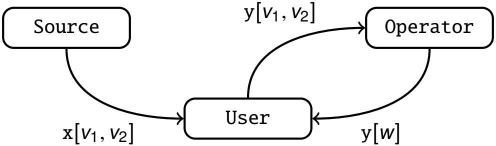
Contracts for choices

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
```

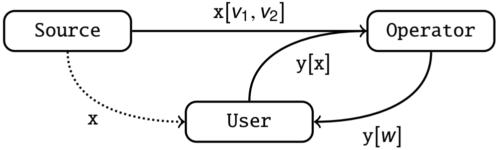
First order interaction and blame

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



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)