Session types for linear multithreaded functional programming

Paper by V. Vasconcellos, A. Ravara & S. Gay
Presentation by H. U. Flemberg, S. S. Nielsen & A. S. Al-Sibahi
Narrated by A. S. Al-Sibahi
Outline

- Introduction
- Language
- Delegation
- Channel Aliasing
- Criticism
Objective

Integration of session type programming in a functional language
Key achievements

- Working implementation in a multi-threaded functional language
  - At the time, only theoretical extensions using π-calculus existed
  - Full static-checking of both ordinary types and session types
- Formalization of semantics
  - Shows that session types can be blended in with ordinary types, in almost orthogonal fashion
Language - intro

S = ?Int.?Int.!Int. End
server u = let x = receive u in
  let y = receive u in
  send x + y on u

S' = !Int.!Int.?Int. End
client u = send 2 on u
  send 3 on u
  let x = receive u in code
server u = case u of {
    add ⇒ send (receive u)+(receive u) on u
    neg ⇒ send −(receive u) on u
}

S’ = Ω (add: !Int.!Int.?Int.End, neg: !Int.?Int.End)
negClient u = select neg on u
    send 2 on u
    let x = receive u in code
Language - functions

\[ S = \& (\text{add}: \ldots, \text{neg}: \ldots, \text{eval}: ?(\text{Int} \rightarrow \text{Bool}).?\text{Int}!.\text{Bool}.\text{End}) \]

server \( u \) = case \( u \) of {
    \( \text{add} \Rightarrow \ldots, \text{neg} \Rightarrow \ldots, \)
    \( \text{eval} \Rightarrow \text{send} (\text{receive} \ u)(\text{receive} \ u) \) on \( u \)
}

\[ S' = \oplus (\text{add}: \ldots, \text{neg}: \ldots, \text{eval}: !(\text{Int} \rightarrow \text{Bool}).!\text{Int}.?\text{Bool}.\text{End}) \]

primeClient \( u \) = select eval on \( u \)
    send isPrime on \( u \)
    send 9007199254740881 on \( u \)
    let \( x = \text{receive} \ u \) in code
Language - ordinary types

\[
\Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma'' \quad \Gamma, x : T \vdash \Sigma'' \triangleright t : U \triangleleft \Sigma'
\]

\[
\Gamma \vdash \Sigma \triangleright \text{let } x = e \text{ in } t : U \triangleleft \Sigma'
\]

\[
\Gamma \vdash v : \text{Bool} \quad \Gamma \vdash \Sigma \triangleright e : T \triangleleft \Sigma' \quad \Gamma \vdash \Sigma \triangleright e' : T \triangleleft \Sigma'
\]

\[
\Gamma \vdash \Sigma \triangleright \text{if } v \text{ then } e \text{ else } e' : T \triangleleft \Sigma'
\]

\[
\Gamma \vdash v : (\Sigma; T \rightarrow U; \Sigma') \quad \Gamma \vdash v' : T
\]

\[
\Gamma \vdash \Sigma \triangleright vv' : U \triangleleft \Sigma'
\]
Language - session types

\[ \Gamma \vdash v : \text{Chan } c \]
\[ \Gamma \vdash \Sigma, c : ?D.S \triangleright \text{receive } v : D \triangleleft \Sigma, c : S \]

\[ \Gamma \vdash v : D \quad \Gamma \vdash v' : \text{Chan } c \]
\[ \Gamma \vdash \Sigma, c : !D.S \triangleright \text{send } v \text{ on } v' : \text{Unit } \triangleleft \Sigma, c : S \]
Delegation

askNeg :: [(add : . . .)] → [S] → Unit
askNeg x y = let u = request x in
            select neg on u; send 7 on u
            let w = request y in
            send u on w; close w

getNeg :: [S] → Unit
getNeg y = let w = accept y in
            let u = receive w in
            let i = receive u in
            close u; close w; code

● Client A establishes connection with server
● Selects neg operation and sends the argument
● Client A connects with Client B
● ...and sends the server channel

● Client B accepts the connection from A
● Receives the server channel
● Receives the negated argument on the server channel
Delegation

\[ \Gamma \vdash v : \text{Chan } d \quad \Gamma \vdash v' : \text{Chan } c \]
\[ \Gamma \vdash \Sigma, c : \text{!}_S' . S, d : S' \triangleright \text{send } v \text{ on } v' : \text{Unit } \triangleleft \Sigma, c : S \]

\[ \Gamma \vdash v : \text{Chan } c \]
\[ \Gamma \vdash \Sigma, c : ?S' . S \triangleright \text{receive } v : \text{Chan } d \triangleleft (\Sigma \bullet d : S'), c : S \]
Channel Aliasing

\[
\text{sendSend } u \; v = \text{send 1 on } u; \; \text{send 2 on } v
\]

- Hard to decide a specific type
  - \( u \equiv v : !\text{Int.}!\text{Int. End} \) instead of \( u : !\text{Int. End} \) and \( v : !\text{Int. End} \)

- Type of function decided at use-site
- Could be solved by enforcing 100% linearity
Channel Aliasing

Example

\[\text{sendSend } u \; v = \text{send} \; 1 \; \text{on} \; u; \; \text{send} \; 2 \; \text{on} \; v\]

\[\Sigma, \; u:!\text{Int.}!\text{Int.}.\text{End}\]

\[\text{client } u = \text{sendSend } u \; u\]

\[\Sigma, \; u:!\text{Int.}.\text{End}, \; v:!\text{Int.}.\text{End}\]

\[\text{client’ } = \text{let } a = \text{new in}\]
\[\quad \text{let } u = \text{request } a \; \text{in}\]
\[\quad \text{let } v = \text{request } a \; \text{in}\]
\[\quad \text{sendSend } u \; v\]
Criticism

- Macro-like functions seems like a hacky solution to the aliasing problem
- Lack of recursive session types
- No external interaction (e.g. networking)
- Lack of evaluation