Haskell Session Types with (Almost) No Class

Presented by Christian Harrington, Thomas Didriksen, Morten Fangel & Sune Alkærsig
Introduction

- Session types implemented in Haskell utilizing the Haskell type system

- Differences from session types in Java
  - No separate protocol specification required
  - Protocols are verified statically by the Haskell type checker
Basic Session Type Primitives

- **Send:** `data (:! :) a r`
- **Receive:** `data (:? :) a r`
- **Choice:** `data (:+ :) a r`
- **Offer:** `data (:& :) a r`
- **End:** `data Eps`

- No constructors, no runtime representation
Basic Session Type Primitives

- Haskell:
  \[ \text{Integer} :!: \text{Bool} :?: \text{Eps} \]

- Session Java:
  \[ !<\text{int}>.?<\text{bool}> \]
Alternation

- **Offer:**
  \[(\text{Integer} : ?: \text{Eps}) &: (\text{Bool} : ?: \text{Eps})]\n
- **Choose:**
  \[(\text{Integer} : !: \text{Eps}) :+: (\text{Bool} : !: \text{Eps})]\n
- Unlike Session Java, offers and choices are binary.
example :: (Int :?: Int :!: Eps)
    &: 
        (String :?: String :!: Eps)
example = offer -- Offer a choice
    (ixdo -- ?<Int>.!<Int>
        num <- recv
        send (-num))
    (ixdo -- ?<String>.!<String>
        str <- recv
        send (reverse str))
Duality

- Implemented using multi-parameter type classes and functional dependencies

```haskell
class Dual r s | r -> s, s -> r

instance Dual Eps Eps
  (D-EPS)

  ε ⨳ ε
```
Duality

instance Dual r s => Dual (a :!: r) (a :?: s)
instance Dual (Dual r1 s1, Dual r2 s2) =>
  Dual (r1 :+: r2) (s1 :&: s2)

(D-CHOOSE)
\[
\begin{align*}
\tau_1 & \bowtie \tau_1' \\
\tau_2 & \bowtie \tau_2'
\end{align*}
\]
\[
\tau_1 \oplus \tau_2 \bowtie \tau_1' \& \tau_2'
\]
Sessions
Session Representation

- **Session type. Evolves a session s to s':**

  ```haskell
  newtype Session s s' a = Session { unSession :: UChan -> IO a }
  ```

- **Phantom parameters (s s')**
  - These parameters are only used for typing, and are not used for construction.
Capabilities

- Represents the capability to run protocol $r$ in session type environment $e$
  
  data Cap e r

- Session type environments are needed for recursion, which will be discussed later
Send/receive

```
send :: a -> Session (Cap e (a :!: r)) (Cap e r) ()
send x = Session (\c -> unsafeWriteUChan c x)
```

```
recv :: Session (Cap e (a :?: r)) (Cap e r) a
recv = Session unsafeReadUChan
```
We need to compose sessions:

- A session from \( s \) to \( s' \) can be composed with one from \( t \) to \( t' \) if \( s' = t \)

This can be represented with indexed monads

An indexed monad \( m \mathbf{i} \mathbf{j} \mathbf{a} \) is parameterized by precondition \( \mathbf{i} \) and postcondition \( \mathbf{j} \), with return type \( \mathbf{a} \).
Indexed monads

class IxMonad m where
  (>>>=) :: m i j a -> (a -> m j k b) -> m i k b
  (>>>>) :: m i j a -> m j k b -> m i k b
  m >>>> k = m >>>= _ -> k
Indexed monads

Operation 1: from \(?\text{int}\).!\text{int}\) to \(!\text{int}\)

\[
op1 :: \text{Session} (\text{Cap e (Int ?:: Int !:: Eps)}) (\text{Cap e (Int !:: Eps)}) \text{Int}
\]

Operation 2: from \(!\text{int}\) to ()

\[
op2 :: \text{Session} (\text{Cap e (Int !:: Eps)}) (\text{Cap e Eps}) ()
\]

Indexed monads allows chaining operation 1 with 2, but not operation 2 with 1.

\[
\text{op1} \gggg \text{op2} \text{ works} \quad \text{op2} \gggg \text{op1} \text{ fails}
\]
Recursion
Recursion

- Represented by
  ```
  data Rec r
  data Var v
  ```

- Example of recursive type
  ```
  Request :!: Rec ((Response :?: Var Z) &: Eps)
  ```
Recursion

- **Entering a recursion**
  enter :: Session (Cap e (Rec r)) (Cap (r, e) r) ()
  enter = Session (\_ \_ -> return ())

- **Repeating**
  zero :: Session (Cap (r, e) (Var Z))
  (Cap (r, e) r) ()
  zero = Session (\_ \_ -> return ())

  suc :: Session (Cap (r, e) (Var (S v)))
  (Cap e (Var v)) ()
  suc = Session (\_ \_ -> return ())
Recursion

server = enter >>> loop where
    loop = offer close
        (ixdo s <- recv
            io (putStrLn s)
            zero
            loop)

- Server’s session type
  Rec (Eps :&: (String :?: Var Z))
Bringing It All Together
newtype Rendezvous r =
  Rendezvous (TChan UChan)

- Synchronization Object
- Phantom Parameter indicating the protocol
- Wraps a typed channel
- Used for establishing a connection with accept and request
Accept

- Requires a **Rendezvous** and a **Session** with matching protocols
- Creates a new channel, sends this channel, and runs the session

```haskell
accept :: Rendezvous r ->
    Session (Cap () r) () a -> IO a
accept (Rendezvous c) (Session f) = do
    nc <- newUChan
    writeTChan c nc
    f nc
```
Request

- Requires a **Rendezvous and a Session** with dual protocols
- Runs the session computation on the given channel form **accept**

```haskell
request :: Dual r r' => Rendezvous r -> Session (Cap () r') () a -> IO a
request (Rendezvous c) (Session f) = readTChan c >>= f
```
Example
Example

server1 ::
  (Integer :: Integer :: Integer :: Eps) &:
  (Integer :: Integer :: Eps)
server1 = offer
  (ixdo a <- recv
   b <- recv
   send (a + b)
   close)
  (ixdo a <- recv
   send (-a)
   close)
Example, continued

server2 :: Eps :&:
    (Integer ::= Integer ::! Eps)
server2 = offer
    close
    (ixdo
        a ← recv
        send (−a)
        close)

- Both have the same protocol for their second option, but different protocols for their first
Example, continued

```
client :: a -> r :+: (a :!: b :?: Eps)
client x = ixdo sel2
  send x
  y <- recv
  close
  ret y

- Functional dependencies can infer that client can be unified with the dual of both Server1 and Server2's protocol
- Client can communicate with both
```
Multiple Channels & Delegation
Multiple Channels

- Channels and capabilities are tied together with a unique tag \( t \)
  
  ```haskell
  newtype Channel t = Channel UChan
  data Cap t e r
  ```

- Session is indexed by a stack of capabilities
  
  ```haskell
  send :: Channel t -> a ->
  Session (Cap t e (a :!: r), x)
  (Cap t e r, x) ()
  send (Channel c) a =
  Session (unsafeWriteUChan c a)
  ```
Delegation

- Implemented purely in the type system, as capabilities have no run-time existence

- Because capabilities do not exist at run-time, unit is the data sent on the channel
Delegation

send_cap :: Channel t -> Session (Cap t e (Cap t' e' r' :!: r),
                          (Cap t' e' r', x))
                          (Cap t e r, x)
                          ()

send_cap (Channel c) = Session (unsafeWriteUChan c ())
Delegation

\[
\text{recv\_cap} :: \text{Channel} \ t \rightarrow \\
\text{Session} \ (\text{Cap} \ t \ e \ (\text{Cap} \ t' \ e' \ r' ::? : r), \ x) \\
(\text{Cap} \ t \ e \ r, \ (\text{Cap} \ t' \ e' \ r', \ x)) \\
()
\]

\[
\text{recv\_cap} \ (\text{Channel} \ c) = \\
\text{Session} \ (\text{unsafeReadUChan} \ c)
\]
Conclusion
Criticism

- Not usable over a network

- Approach is applicable to other languages, but might not be ideal

- De Bruijn indices are not user friendly
  - Their alternative solutions have other drawbacks
Conclusion

- Inference of session types is cool
  - You do not have to write a session specification

- Almost everything can be checked statically by the Haskell type checker