An introduction to Session Types

by Wen Kokke

Dramatis Personæ

- channel
  a tube to send messages through
- endpoint
  either end of a channel
- session
  series of messages sent over one channel

Session Types at a glance

```haskell
server :: Recv RFC (Send (Either Cake Nope) End) -> IO ()
server c = do
    (msg, c') <- recv c
    case msg of
        "May I have cake, please?" -> do c'' <- send (Left 🎂) c'; close c''
        "May I have cake?"         -> do c'' <- send (Right 🙌) c'; close c''

client :: Send RFC (Recv (Maybe Cake) End) -> IO Mood
client c = do
    c' <- send "May I have cake, please?" c
    (resp, c'') <- recv c'
    wait c''
    case resp of
       Left 🎂 -> return 😍
       Right 🙌 -> return 😭
```

Session Types at a glance

```
server :: ?RFC.!(Either Cake Nope).End -> IO ()
server c = do
    (msg, c') <- recv c
    case msg of
        "May I have cake, please?" -> do c'' <- send (Left 🎂) c'; close c''
        "May I have cake?"         -> do c'' <- send (Right 🙌) c'; close c''

client :: !RFC.?(Either Cake Nope).End -> IO Mood
client c = do
    c' <- send "May I have cake, please?" c
    (resp, c'') <- recv c'
    wait c''
    case resp of
       Left 🎂 -> return 😍
       Right 🙌 -> return 😭
```

Roadmap

- The untyped λ-calculus! (So powerful, so scary…)
- Taming the λ-calculus with types…

- The untyped π-calculus! (Is even scarier…)
- Taming the π-calculus with types…

- Concurrent λ-calculus! (λ and π together…)

The untyped lambda calculus

$$\begin{array}{l} \text{Term } L, M, N \\ \quad ::= \quad x \mid \lambda x.\, M \mid M\ N \end{array}$$

$$(\lambda x.\, M)\ N \longrightarrow M\{N/x\}$$

$$\frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N} \qquad \frac{N \longrightarrow N'}{M\ N \longrightarrow M\ N'}$$

The untyped lambda calculus is powerful

$$
\begin{aligned}
Y &\quad = \quad \lambda f.\, (\lambda x.\, x\ x)(\lambda x.\, f\ (x\ x)) \\[2em]
Y\ f &\quad \longrightarrow \quad f\ (Y\ f) \\
&\quad \longrightarrow \quad f\ (f\ (Y\ f)) \\
&\quad \longrightarrow \quad f\ (f\ (f\ (Y\ f))) \\
&\quad \longrightarrow \quad \ldots
\end{aligned}
$$

The untyped lambda calculus is scary

$$
\begin{aligned}
Y &= \lambda f.\, (\lambda x.\, x\ x)(\lambda x.\, f\ (x\ x)) \\
\\
\Omega &= (\lambda x.\, x\ x)(\lambda x.\, x\ x) \\
&\longrightarrow (x\ x)\{\lambda x.\, x\ x / x\} \\
&= (\lambda x.\, x\ x)(\lambda x.\, x\ x)
\end{aligned}
$$

The untyped lambda calculus is scary

If we want <u>more</u> than just functions...

$$\mathit{plus}\ 1\ \mathit{true}$$

...we have to worry about silly stuff like this!

Taming the lambda calculus with types

$$\frac{\Gamma \ni x : A}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

Taming the lambda calculus with types

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \multimap B}$$

$$\frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash M\, N : B}$$

Let's talk pi calculus

The untyped pi calculus syntax

$$
\begin{array}{llll}
\text{Process } P, Q, R & & & \\
& ::= & (\nu x)P & \text{— create new channel} \\
& \mid & (P \parallel Q) & \text{— put } P \text{ and } Q \text{ in parallel} \\
& \mid & 0 & \text{— done} \\
& \mid & x\langle y \rangle.\, P & \text{— send } y \text{ on } x \\
& \mid & x(y).\, P & \text{— receive } y \text{ on } x \\
& \mid & !P & \text{— replicate } P
\end{array}
$$

The untyped pi calculus semantics

$$(\nu x)(x\langle y\rangle.\,P \parallel x(z).\,Q) \longrightarrow (\nu x)(P \parallel Q\{y/z\})$$

$$\frac{P \longrightarrow P'}{(\nu x)P \longrightarrow (\nu x)P'}$$

$$\frac{P \longrightarrow P'}{P \parallel Q \longrightarrow P' \parallel Q} \qquad \frac{Q \longrightarrow Q'}{P \parallel Q \longrightarrow P \parallel Q'}$$

The untyped pi calculus semantics

How do we reduce...?

$$(\nu x)(x(z).\,Q \parallel x\langle y\rangle.\,P)$$

Maybe we can apply...?

$$(\nu x)(x\langle y\rangle.\,P \parallel x(z).\,Q) \longrightarrow (\nu x)(P \parallel Q\{y/z\})$$

Nope!

The untyped pi calculus semantics

$$
\begin{array}{lcll}
P \parallel Q & \equiv & Q \parallel P & \\
P \parallel (Q \parallel R) & \equiv & (P \parallel Q) \parallel R & \\
P \parallel 0 & \equiv & P & \\
(\nu x)(\nu y)P & \equiv & (\nu y)(\nu x)P & \\
(\nu x)(P \parallel Q) & \equiv & (\nu x)P \parallel Q, & \text{if } x \notin Q \\
!P & \equiv & !P \parallel P &
\end{array}
$$

$$
\frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}
$$

The untyped pi calculus semantics

$$\frac{(\nu x)(x(z).\,Q \parallel x\langle y\rangle.\,P) \equiv (\nu x)(x\langle y\rangle.\,P \parallel x(z).\,Q) \qquad (\nu x)(x\langle y\rangle.\,P \parallel x(z).\,Q) \longrightarrow (\nu x)(P \parallel Q\{y/z\}) \qquad (\nu x)(P \parallel Q\{y/z\}) \equiv (\nu x)(Q\{y/z\} \parallel P)}{(\nu x)(x(z).\,Q \parallel x\langle y\rangle.\,P) \longrightarrow (\nu x)(Q\{y/z\} \parallel P)}$$

The untyped pi calculus is scary

$$(\nu x)\begin{pmatrix} x\langle y_1\rangle.\,P_1 \parallel x(z_1).\,Q_1 \parallel \\ x\langle y_2\rangle.\,P_2 \parallel x(z_2).\,Q_2 \end{pmatrix}$$

$$\downarrow$$

$$(\nu x)\begin{pmatrix} P_1 \parallel Q_1\{y_1/z_1\} \parallel \\ P_2 \parallel Q_2\{y_2/z_2\} \end{pmatrix} \quad or \quad (\nu x)\begin{pmatrix} P_1 \parallel Q_1\{y_2/z_1\} \parallel \\ P_2 \parallel Q_2\{y_1/z_2\} \end{pmatrix}$$

The untyped pi calculus is scary

$$(\nu x)\,(x(z).\,P \parallel x(w).\,Q)$$

$$\downarrow$$

✨ *nothing* ✨

Taming the pi calculus with types

Process P, Q, R

$$\begin{array}{rll} ::= & (\nu x x')P & \text{— create new channel } x{\leftrightarrow}x' \\ \mid & \ldots & \end{array}$$

Session type S

$$\begin{array}{rll} ::= & !S.\,S' & \text{— send} \\ \mid & ?S.\,S' & \text{— receive} \\ \mid & \mathbf{end} & \text{— done} \end{array}$$

Duality

$$\begin{array}{rcl} \overline{!S.\,S'} & = & ?S.\,\overline{S'} \\ \overline{?S.\,S'} & = & !S.\,\overline{S'} \\ \overline{\mathbf{end}} & = & \mathbf{end} \end{array}$$

Taming the pi calculus with types

$$\frac{\Gamma, x : S, x' : \overline{S} \vdash P}{\Gamma \vdash (\nu x x')P} \qquad \frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma, \Delta \vdash P \parallel Q}$$

$$\frac{}{\varnothing \vdash \mathbf{0}} \qquad \frac{\Gamma \vdash P}{\Gamma, x : \mathbf{end} \vdash P}$$

$$\frac{\Gamma, x : B \vdash P}{\Gamma, x : !A.\, B, y : A \vdash x\langle y\rangle.\, P} \qquad \frac{\Gamma, y : A, x : B \vdash P}{\Gamma, x : ?A.\, B \vdash x(y).\, P}$$

Taming the pi calculus too much

we can't do choice

😭

Taming the pi calculus too little

$$(\nu x x')(\nu y y')\,(x(z).\,y'\langle z\rangle.\,P \parallel y(w).\,x'\langle w\rangle.\,Q)$$

$$\downarrow$$

✨ *nothing* ✨

Concurrent lambda calculus

λ 💥 π

Concurrent lambda calculus

Term L, M, N		Process P, Q, R	
$::=$	x	$::=$	$(\nu x x')P$
$\mid$	$\lambda x.\, M$	$\mid$	$(P \parallel Q)$
$\mid$	$M\ N$	$\mid$	0
		$\mid$	$x\langle y\rangle.\, P$
		$\mid$	$x(y).\, P$

Concurrent lambda calculus

$$
\begin{array}{rcl}
\text{Term } L, M, N & ::= & x \\
& \mid & \lambda x.\, M \\
& \mid & M\ N \\
& \mid & K
\end{array}
\qquad
\begin{array}{rcl}
\text{Process } P, Q, R & ::= & (\nu x x')P \\
& \mid & (P \parallel Q) \\
& \mid & M
\end{array}
$$

$$\text{Const } K ::= \mathbf{send} \mid \mathbf{recv}$$

Concurrent lambda calculus

$$
\begin{array}{rcl}
\text{Term } L, M, N & & \\
::= & x & \\
| & \lambda x.\, M & \\
| & M\ N & \\
| & K &
\end{array}
\qquad
\begin{array}{rcl}
\text{Process } P, Q, R & & \\
::= & (\nu x x')P & \\
| & (P \parallel Q) & \\
| & M &
\end{array}
$$

$$\text{Const } K ::= \textbf{send} \mid \textbf{recv} \mid \textbf{new} \mid \textbf{spawn}$$

Session Types at a glance

```
server :: ?RFC.!(Either Cake Nope).End -> IO ()
server c = do
    (msg, c') <- recv c
    case msg of
        "May I have cake, please?" -> do c'' <- send (Left 🎂) c'; close c''
        "May I have cake?"         -> do c'' <- send (Right 🙌) c'; close c''

client :: !RFC.?(Either Cake Nope).End -> IO Mood
client c = do
    c' <- send "May I have cake, please?" c
    (resp, c'') <- recv c'
    wait c''
    case resp of
      Left 🎂 -> return 😍
      Right 🙌 -> return 😭
```

Concurrent lambda calculus is still unsafe

$$(\nu x x')(\nu y y')\left(\begin{array}{l}\mathbf{let}\ (\_, z) = \mathbf{recv}\ x\ \mathbf{in}\ \mathbf{send}\ z\ y; M \\ \mathbf{let}\ (\_, w) = \mathbf{recv}\ x'\ \mathbf{in}\ \mathbf{send}\ w\ y'; N\end{array}\right)$$

$$\downarrow$$

✨ *nothing* ✨

Where do we go from here?

Deny deadlocks?

- acyclic communication graphs
- priorities and global deadlock freedom

Where do we go from here?

Controlled non-determinism?

- non-deterministic local choice
- guarded global choice
- shared channels
- ...

Roadmap

- Taming the λ-calculus
  Recursion and termination
- Taming the π-calculus
  Concurrent λ-calculus
  Deadlock freedom
  Controlled non-determinism