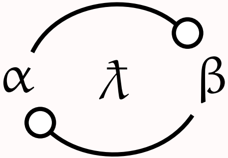


A Self-Dual Distillation of Session Types

(Functional Pearl)

Jules Jacobs

Radboud University Nijmegen
mail@julesjacobs.com

Usual message passing:

- Stream of messages of fixed type
- e.g., Go, Rust

Usual message passing:

- Stream of messages of fixed type
- e.g., Go, Rust

Session types:

- Flexible message passing protocols
- Type of message can depend on the state of the protocol

Two flavours

π **calculus: “everything is a channel”**

- Elegant minimalist session types
- Kobayashi 2002, Dardha et al. 2012, Arslanagic et al. 2019

Two flavours

π calculus: “everything is a channel”

- Elegant minimalist session types
- Kobayashi 2002, Dardha et al. 2012, Arslanagic et al. 2019

λ calculus: “everything is a function”

- GV = linear λ calculus + channels + session types

Two flavours

π calculus: “everything is a channel”

- Elegant minimalist session types
- Kobayashi 2002, Dardha et al. 2012, Arslanagic et al. 2019

λ calculus: “everything is a function”

- GV = linear λ calculus + channels + session types
- Not aiming at a minimalist concurrent calculus

Two flavours

π calculus: “everything is a channel”

- Elegant minimalist session types
- Kobayashi 2002, Dardha et al. 2012, Arslanagic et al. 2019

λ calculus: “everything is a function”

- GV = linear λ calculus + channels + session types
- Not aiming at a minimalist concurrent calculus

This work: ƛ calculus = λ calculus + barriers

- Minimal concurrent extension of linear λ calculus
- Only one new operation:
  fork : $((\alpha \multimap \beta) \multimap \mathbf{1}) \multimap (\beta \multimap \alpha)$
- Everything is a function
- Session types as function types
- Simpler meta theory

Session types 101: GV (Gay, Vasconcelos, Wadler, . . .)

```
let c′ = fork(λc.
        let (c, n) = receive(c) in
        let c = send(c, n mod 2 ≡ 0) in
        close(c))
let c′ = send(c′, 3) in
let (c′, msg) = receive(c′) in
close(c′)
```

Session types 101: GV (Gay, Vasconcelos, Wadler, . . .)

```
let c′ : !Int. ?Bool. End = fork(λc : ?Int. !Bool. End.
        let (c, n) = receive(c) in
        let c = send(c, n mod 2 ≡ 0) in
        close(c))
let c′ = send(c′, 3) in
let (c′, msg) = receive(c′) in
close(c′)
```

Session types 101: GV (Gay, Vasconcelos, Wadler, . . .)

```
let c′ : !Int. ?Bool. End = fork(λc : ?Int. !Bool. End.
        let (c : !Bool. End, n : Int) = receive(c) in
        let c = send(c,  n mod 2 ≡ 0) in
        close(c))
let c′ : ?Bool. End = send(c′, 3) in
let (c′, msg) = receive(c′) in
close(c′)
```

Session types 101: GV (Gay, Vasconcelos, Wadler, . . .)

```
let c′ : !Int. ?Bool. End = fork(λc : ?Int. !Bool. End.
        let (c : !Bool. End, n : Int) = receive(c) in
        let c : End = send(c,  n mod 2 ≡ 0) in
        close(c))
let c′ : ?Bool. End = send(c′, 3) in
let (c′ : End, msg : Bool) = receive(c′) in
close(c′)
```

Session types 101: GV (Gay, Vasconcelos, Wadler, . . .)

Linear λ calculus: $\tau ::= \mathbf{0} \mid \mathbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau$

Session types: $s ::= \mathbf{!}\tau.s \mid \mathbf{?}\tau.s \mid s \oplus s \mid s \mathbin{\&} s \mid \mathsf{End}$

send : $(\mathbf{!}\tau.s) \times \tau \multimap s$

receive : $(\mathbf{?}\tau.s) \multimap (s \times \tau)$

tell$_\mathbf{L}$: $(s_1 \oplus s_2) \multimap s_1$

tell$_\mathbf{R}$: $(s_1 \oplus s_2) \multimap s_2$

ask : $(s_1 \mathbin{\&} s_2) \multimap (s_1 + s_2)$

close : $\mathsf{End} \multimap \mathbf{1}$

fork : $(s \multimap \mathbf{1}) \multimap \overline{s}$

$$\overline{\mathbf{!}\tau.s} \triangleq \mathbf{?}\tau.\overline{s}$$

$$\overline{\mathbf{?}\tau.s} \triangleq \mathbf{!}\tau.\overline{s}$$

$$\overline{s_1 \oplus s_2} \triangleq \overline{s_1} \mathbin{\&} \overline{s_2}$$

$$\overline{s_1 \mathbin{\&} s_2} \triangleq \overline{s_1} \oplus \overline{s_2}$$

$$\overline{\mathsf{End}} \triangleq \mathsf{End}$$

“Session types” 101: ƛ

Linear λ calculus: $\tau ::= \mathbf{0} \mid \mathbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau$

“Session types” 101: ƛ

Linear λ calculus: $\tau ::= \mathbf{0} \mid \mathbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau$

$$\mathbf{fork} : ((\alpha \multimap \beta) \multimap \mathbf{1}) \multimap (\beta \multimap \alpha)$$

“Session types” 101: $\bar{\lambda}$

Linear λ calculus: $\tau ::= \mathbf{0} \mid \mathbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau$

$$\textbf{fork} : ((\alpha \multimap \beta) \multimap \mathbf{1}) \multimap (\beta \multimap \alpha)$$

That’s it!

"Session types" 101: $\hbar$

Linear λ calculus: $\tau ::= \mathbf{0} \mid \mathbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau$

$$\textbf{fork} : ((\alpha \multimap \beta) \multimap \mathbf{1}) \multimap (\beta \multimap \alpha)$$

That's it!

One new operation, no new types, no dual $\overline{!\tau.s}$

“Session types” 101: $\lambda\!\!\!\!-$

Linear λ calculus: $\tau ::= \mathbf{0} \mid \mathbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau$

$$\mathbf{fork} : ((\alpha \multimap \beta) \multimap \mathbf{1}) \multimap (\beta \multimap \alpha)$$

That’s it!

One new operation, no new types, no dual $\overline{!\tau.s}$

$$\mathbf{fork}(\lambda x.\, E_1)$$

“Session types” 101: ƛ

Linear λ calculus: $\tau ::= \mathbf{0} \mid \mathbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau$

$$\mathbf{fork} : ((\alpha \multimap \beta) \multimap \mathbf{1}) \multimap (\beta \multimap \alpha)$$

That’s it!

One new operation, no new types, no dual $\overline{!\tau.s}$

$$\mathbf{fork}(\lambda \underset{\boxed{\alpha \multimap \beta}}{x}.\, E_1)$$

“Session types” 101: ƛ

Linear λ calculus: $\tau ::= \mathbf{0} \mid \mathbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau$

$$\mathbf{fork} : ((\alpha \multimap \beta) \multimap \mathbf{1}) \multimap (\beta \multimap \alpha)$$

That’s it!

One new operation, no new types, no dual $\overline{!\tau.s}$

$$\mathbf{let}\ x' = \mathbf{fork}(\lambda x.\, E_1)\ \mathbf{in}\ E_2$$

$$\boxed{\alpha \multimap \beta}$$

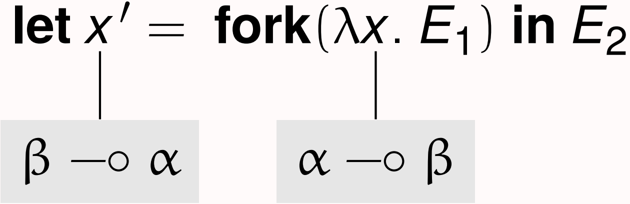
"Session types" 101: ƛ

Linear λ calculus: $\tau ::= \mathbf{0} \mid \mathbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau$

$$\mathbf{fork} : ((\alpha \multimap \beta) \multimap \mathbf{1}) \multimap (\beta \multimap \alpha)$$

That's it!

One new operation, no new types, no dual $\overline{!\tau.s}$



$$\mathbf{let}\ x' = \mathbf{fork}(\lambda x.\ E_1)\ \mathbf{in}\ E_2$$

$\beta \multimap \alpha$

$\alpha \multimap \beta$

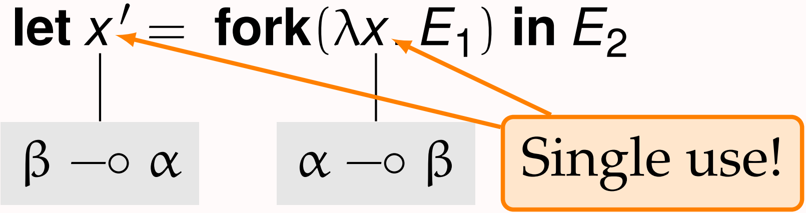
“Session types” 101: ƛ

Linear λ calculus: $\tau ::= \mathbf{0} \mid \mathbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau$

$$\mathbf{fork} : ((\alpha \multimap \beta) \multimap \mathbf{1}) \multimap (\beta \multimap \alpha)$$

That’s it!

One new operation, no new types, no dual $\overline{!\tau.s}$



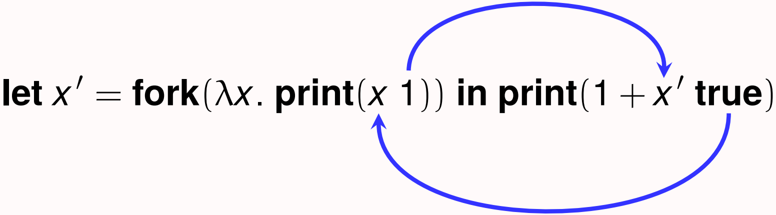
$$\mathbf{let}\ x' = \mathbf{fork}(\lambda x.\, E_1)\ \mathbf{in}\ E_2$$

$\beta \multimap \alpha$

$\alpha \multimap \beta$

Single use!

$$\textbf{let}\ x' = \textbf{fork}(\lambda x.\ \textbf{print}(x\ 1))\ \textbf{in}\ \textbf{print}(1 + x'\ \textbf{true})$$



$$\textbf{let}\ x' = \textbf{fork}(\lambda x.\ \textbf{print}(x\ 1))\ \textbf{in}\ \textbf{print}(1 + x'\ \textbf{true})$$

$$\textbf{let}\ x' = \textbf{fork}(\lambda x.\ \textbf{print}(\textbf{true}))\ \textbf{in}\ \textbf{print}(1+1)$$

Operational semantics

$$\rho \in \mathsf{N} \xrightharpoonup{\text{fin}} \mathsf{Thread}(e) \mid \mathsf{Barrier}$$

Operational semantics

$$\rho \in \mathsf{N} \xrightarrow{\mathsf{fin}} \mathsf{Thread}(e) \mid \mathsf{Barrier}$$

$$\{n \mapsto \mathsf{Thread}(K[\,e_1\,])\} \rightsquigarrow \{n \mapsto \mathsf{Thread}(K[\,e_2\,])\} \quad \text{if } e_1 \rightsquigarrow_{\text{pure}} e_2$$

$$\{n \mapsto \mathsf{Thread}(K[\,\mathbf{fork}(v)\,])\} \rightsquigarrow \begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\;\langle k\rangle\;]) \\ k \mapsto \mathsf{Barrier} \\ m \mapsto \mathsf{Thread}(v\;\;\langle k\rangle) \end{Bmatrix} \qquad (\textbf{fork})$$

$$\begin{Bmatrix} n \mapsto \mathsf{Thread}(K_1[\;\langle k\rangle\;\;v_1\,]) \\ k \mapsto \mathsf{Barrier} \\ m \mapsto \mathsf{Thread}(K_2[\;\langle k\rangle\;\;v_2\,]) \end{Bmatrix} \rightsquigarrow \begin{Bmatrix} n \mapsto \mathsf{Thread}(K_1[\,v_2\,]) \\ m \mapsto \mathsf{Thread}(K_2[\,v_1\,]) \end{Bmatrix} \qquad (\textbf{sync})$$

$$\{n \mapsto \mathsf{Thread}(())\} \rightsquigarrow \{\} \qquad (\text{exit})$$

$$\rho_1 \uplus \rho' \rightsquigarrow \rho_2 \uplus \rho' \quad \text{if } \rho_1 \rightsquigarrow \rho_2 \qquad (\text{frame})$$

$$
\begin{aligned}
&\textbf{let } x' = \textbf{fork}(\lambda x.\ \textbf{let } (y, n) = x\ ()\\
&\qquad\qquad\qquad\qquad\quad \textbf{in } y\ (n \bmod 2 \equiv 0))\\
&\textbf{let } y' = \textbf{fork}(\lambda y.\ x'\ (y, 3))\\
&\textbf{in print}(y'\ ())
\end{aligned}
$$

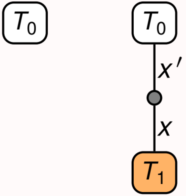
$$\begin{aligned}
&\textbf{let } x' = \textbf{fork}(\lambda x.\, \textbf{let } (y, n) = x\ () \\
&\qquad\qquad\qquad\qquad\qquad \textbf{in } y\ (n \bmod 2 \equiv 0)) \\
&\textbf{let } y' = \textbf{fork}(\lambda y.\, x'\ (y, 3)) \\
&\textbf{in print}(y'\ ())
\end{aligned}$$

$\boxed{T_0}$

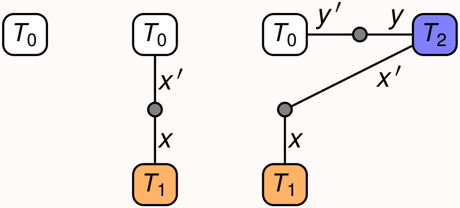
$$\textbf{let}\ x' = \textbf{fork}(\lambda x.\ \textbf{let}\ (y, n) = x\ ()\ \textbf{in}\ y\ (n \bmod 2 \equiv 0))$$

$$\textbf{let}\ y' = \textbf{fork}(\lambda y.\ x'\ (y, 3))$$

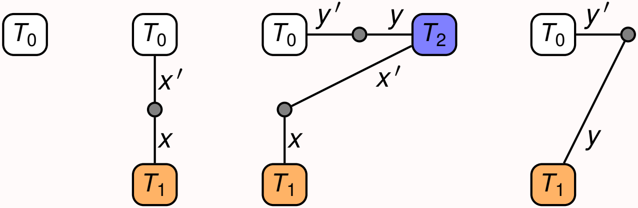
$$\textbf{in print}(y'\ ())$$



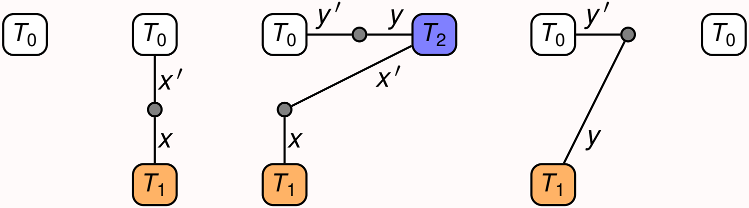
$$\begin{aligned}
&\textbf{let } x' = \textbf{fork}(\lambda x.\ \textbf{let } (y, n) = x\ () \\
&\qquad\qquad\qquad\qquad\quad \textbf{in } y\ (n \bmod 2 \equiv 0)) \\
&\textbf{let } y' = \textbf{fork}(\lambda y.\ x'\ (y, 3)) \\
&\textbf{in print}(y'\ ())
\end{aligned}$$



$$\begin{array}{l}\textbf{let } x' = \textbf{fork}(\lambda x.\ \textbf{let } (y, n) = x\ ()\\ \qquad\qquad\qquad\qquad\qquad \textbf{in } y\ (n \bmod 2 \equiv 0))\\ \textbf{let } y' = \textbf{fork}(\lambda y.\ x'\ (y, 3))\\ \textbf{in print}(y'\ ())\end{array}$$



$$
\begin{aligned}
&\textbf{let } x' = \textbf{fork}(\lambda x.\, \textbf{let } (y, n) = x\ ()\\
&\qquad\qquad\qquad\qquad\quad \textbf{in } y\ (n \bmod 2 \equiv 0))\\
&\textbf{let } y' = \textbf{fork}(\lambda y.\, x'\ (y, 3))\\
&\textbf{in print}(y'\ ())
\end{aligned}
$$



Channel operations as macros

$$\begin{aligned}
\mathbf{fork}_{\mathrm{chan}}(f) &\triangleq \mathbf{fork}(f) \\
\mathbf{send}(c, x) &\triangleq \mathbf{fork}(\lambda c'.\, c\ (c', x)) \\
\mathbf{receive}(c) &\triangleq c\ () \\
\mathbf{close}(c) &\triangleq c\ ()
\end{aligned}$$

Channel operations as macros

$$\begin{aligned}
\textbf{fork}_{\text{chan}}(f) &\triangleq \textbf{fork}(f)\\
\textbf{send}(c, x) &\triangleq \textbf{fork}(\lambda c'.\ c\ (c', x))\\
\textbf{receive}(c) &\triangleq c\ ()\\
\textbf{close}(c) &\triangleq c\ ()
\end{aligned}$$

$$\begin{array}{l}
\textbf{let}\ c' = \textbf{fork}(\lambda c.\\
\qquad \textbf{let}\ (c, n) = \textbf{receive}(c)\ \textbf{in}\\
\qquad \textbf{let}\ c = \textbf{send}(c,\ n \bmod 2 \equiv 0)\ \textbf{in}\\
\qquad \textbf{close}(c))\\
\textbf{let}\ c' = \textbf{send}(c', 3)\ \textbf{in}\\
\textbf{let}\ (c', msg) = \textbf{receive}(c')\ \textbf{in}\\
\textbf{close}(c')
\end{array}$$

Channel operations as macros

$$\begin{aligned}
\mathbf{fork}_{\text{chan}}(f) &\triangleq \mathbf{fork}(f)\\
\mathbf{send}(c, x) &\triangleq \mathbf{fork}(\lambda c'.\ c\ (c', x))\\
\mathbf{receive}(c) &\triangleq c\ ()\\
\mathbf{close}(c) &\triangleq c\ ()
\end{aligned}$$

$\mathbf{let}\ c' : [\![\ \text{!Int.}\ \text{?Bool.}\ \text{End}\ ]\!] = \mathbf{fork}(\lambda c : [\![\ \text{?Int.}\ \text{!Bool.}\ \text{End}\ ]\!].$

$\quad\mathbf{let}\ (c, n) = \mathbf{receive}(c)\ \mathbf{in}$

$\quad\mathbf{let}\ c = \mathbf{send}(c,\ n \bmod 2 \equiv 0)\ \mathbf{in}$

$\quad\mathbf{close}(c))$

$\mathbf{let}\ c' = \mathbf{send}(c', 3)\ \mathbf{in}$

$\mathbf{let}\ (c', msg) = \mathbf{receive}(c')\ \mathbf{in}$

$\mathbf{close}(c')$

Channel operations as macros

$$\begin{aligned}
\textbf{fork}_{\text{chan}}(f) &\triangleq \textbf{fork}(f) \\
\textbf{send}(c, x) &\triangleq \textbf{fork}(\lambda c'.\, c\ (c', x)) \\
\textbf{receive}(c) &\triangleq c\ () \\
\textbf{close}(c) &\triangleq c\ ()
\end{aligned}$$

$$\begin{aligned}
&\textbf{let } c' : [\![\ \textbf{!Int. ?Bool. End}\ ]\!] = \textbf{fork}(\lambda c : [\![\ \textbf{?Int. !Bool. End}\ ]\!]. \\
&\qquad \textbf{let } (c : [\![\ \textbf{!Bool. End}\ ]\!], n : \textbf{Int}) = \textbf{receive}(c) \textbf{ in} \\
&\qquad \textbf{let } c : [\![\ \textbf{End}\ ]\!] = \textbf{send}(c,\ n \bmod 2 \equiv 0) \textbf{ in} \\
&\qquad \textbf{close}(c)) \\
&\textbf{let } c' : [\![\ \textbf{?Bool. End}\ ]\!] = \textbf{send}(c', 3) \textbf{ in} \\
&\textbf{let } (c' : [\![\ \textbf{End}\ ]\!], msg : \textbf{Bool}) = \textbf{receive}(c') \textbf{ in} \\
&\textbf{close}(c')
\end{aligned}$$

Session types as linear function types

$$
\begin{aligned}
[\![\mathsf{End}]\!] &\triangleq \mathbf{1} \multimap \mathbf{1} \\
[\![\mathbf{!}\tau.s]\!] &\triangleq [\![\overline{s}]\!] \times \tau \multimap \mathbf{1} \\
[\![\mathbf{?}\tau.s]\!] &\triangleq \mathbf{1} \multimap [\![s]\!] \times \tau \\
[\![s_1 \oplus s_2]\!], [\![s_1 \mathbin{\&} s_2]\!] &\triangleq \text{(see paper)}
\end{aligned}
$$

Session types as linear function types

$$\begin{aligned}
[\![\mathsf{End}]\!] &\triangleq \mathbf{1} \multimap \mathbf{1} \\
[\![\mathbf{!}\tau.s]\!] &\triangleq [\![\overline{s}]\!] \times \tau \multimap \mathbf{1} \\
[\![\mathbf{?}\tau.s]\!] &\triangleq \mathbf{1} \multimap [\![s]\!] \times \tau \\
[\![s_1 \oplus s_2]\!], [\![s_1 \,\&\, s_2]\!] &\triangleq \text{(see paper)}
\end{aligned}$$

$$\begin{aligned}
\mathbf{fork}_{\text{chan}} : ([\![s]\!] \multimap \mathbf{1}) \multimap [\![\overline{s}]\!] \quad &\triangleq \lambda x.\, \mathbf{fork}(x) \\
\mathbf{close} : [\![\mathsf{End}]\!] \multimap \mathbf{1} \quad &\triangleq \lambda c.\, c\ () \\
\mathbf{send} : [\![\mathbf{!}\tau.s]\!] \times \tau \multimap [\![s]\!] \quad &\triangleq \lambda(c, x).\, \mathbf{fork}(\lambda c'.\, c\ (c', x)) \\
\mathbf{receive} : [\![\mathbf{?}\tau.s]\!] \multimap [\![s]\!] \times \tau \quad &\triangleq \lambda c.\, c\ ()
\end{aligned}$$

Session types as linear function types

$$
\begin{aligned}
[\![\mathsf{End}]\!] &\triangleq \mathbf{1} \multimap \mathbf{1} \\
[\![\mathbf{!}\tau.s]\!] &\triangleq [\![\overline{s}]\!] \times \tau \multimap \mathbf{1} \\
[\![\mathbf{?}\tau.s]\!] &\triangleq \mathbf{1} \multimap [\![s]\!] \times \tau \\
[\![s_1 \oplus s_2]\!], [\![s_1 \mathbin{\&} s_2]\!] &\triangleq \text{(see paper)}
\end{aligned}
$$

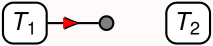
$$
\begin{aligned}
\mathbf{fork}_{\text{chan}} &: ([\![s]\!] \multimap \mathbf{1}) \multimap [\![\overline{s}]\!] && \triangleq \lambda x.\, \mathbf{fork}(x) \\
\mathbf{close} &: [\![\mathsf{End}]\!] \multimap \mathbf{1} && \triangleq \lambda c.\, c\ () \\
\mathbf{send} &: [\![\mathbf{!}\tau.s]\!] \times \tau \multimap [\![s]\!] && \triangleq \lambda(c, x).\, \mathbf{fork}(\lambda c'.\, c\ (c', x)) \\
\mathbf{receive} &: [\![\mathbf{?}\tau.s]\!] \multimap [\![s]\!] \times \tau && \triangleq \lambda c.\, c\ ()
\end{aligned}
$$

Theorem. If GV program is well-typed, then macro expanded ƛ program is well-typed

Theorem. Macro expanded ƛ program simulates GV program

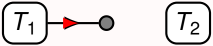
Deadlock freedom: linearity

$\textbf{let } x' = \textbf{fork}(\lambda x.\, ())\ \textbf{in } x'\ 0$ Deadlock!



Deadlock freedom: linearity

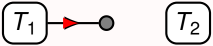
$\textbf{let}\ x' = \textbf{fork}(\lambda x.\ ())\ \textbf{in}\ x'\ 0$ Deadlock!



Ruled out by linear typing

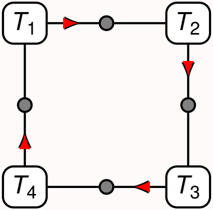
Deadlock freedom: linearity

$\textbf{let}\ x' = \textbf{fork}(\lambda x.\ ())\ \textbf{in}\ x'\ 0$ Deadlock!

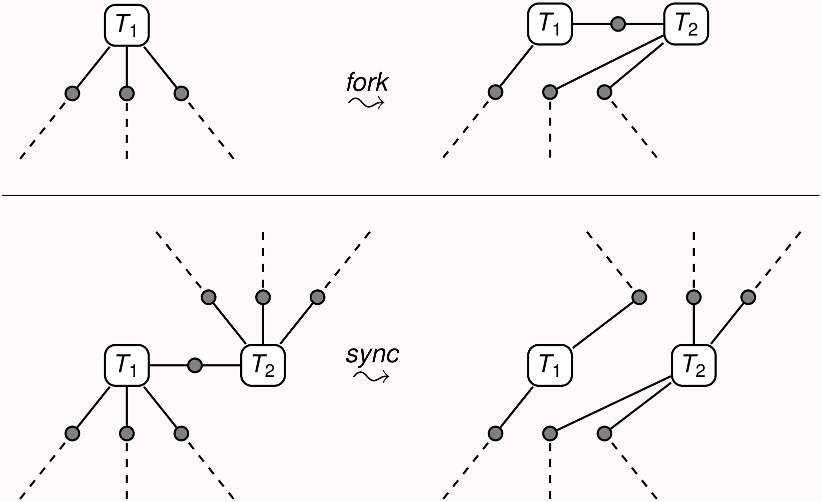


Ruled out by linear typing

But what about cycles?



Deadlock freedom: acyclicity



Mechanized proofs in Coq

Meta theory of λ + recursive types + non-linear types

- Global progress:
  $(e : 1) \wedge \{0 \mapsto e\} \rightsquigarrow \rho \implies \rho$ can step $\vee\ \rho = \{\}$
- Partial deadlock freedom (see paper)
- Memory leak freedom (see paper)

Mechanized proofs in Coq

Meta theory of ƛ + recursive types + non-linear types

- Global progress:
  $(e : 1) \wedge \{0 \mapsto e\} \rightsquigarrow \rho \implies \rho \text{ can step} \vee \rho = \{\}$
- Partial deadlock freedom (see paper)
- Memory leak freedom (see paper)
- Mechanized in Coq (1229 lines)
  - Earlier GV mechanization: 2139 lines
  - (Both use graph library of 5000 lines + Iris/stdpp)

Mechanized proofs in Coq

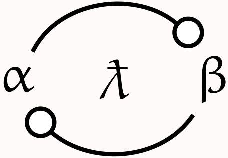
Meta theory of ƛ + recursive types + non-linear types

- Global progress:
  $(e : 1) \wedge \{0 \mapsto e\} \rightsquigarrow \rho \implies \rho \text{ can step} \vee \rho = \{\}$
- Partial deadlock freedom (see paper)
- Memory leak freedom (see paper)
- Mechanized in Coq (1229 lines)
  - Earlier GV mechanization: 2139 lines
  - (Both use graph library of 5000 lines + Iris/stdpp)

Session types in ƛ

- Compiler from GV to ƛ
- Proof that output ƛ program is well-typed
- Proof that output ƛ program simulates GV program
- Mechanized in Coq (568 lines)

$$\underbrace{\textbf{fork} : ((\alpha \multimap \beta) \multimap \mathbf{1}) \multimap (\beta \multimap \alpha)}_{\text{Session types distilled}}$$



Lots of related work and details
in the paper and mechanization

Questions?

mail@julesjacobs.com