Haskell Session Types with (Almost) No Class

Riccardo Pucella   Jesse A. Tov

College of Computer and Information Science
Northeastern University

Haskell Symposium
25 September 2008

Problem: Ordering a Pizza

Client: Hi. What pizza toppings do you have?

Server: We have asparagus, broccoli, cauliflower, . . .

Client: I'd like a medium pizza with olives and mushrooms.

Server: That will be C$12.87. What's your address?

Client: I'm at the Delta Victoria, 45 Songhees Road, Ascot room.

Problem: Ordering a Pizza

Client: Hi. What pizza toppings do you have?
Server: We have asparagus, broccoli, cauliflower, . . .
Client: I'd like a medium pizza with olives and mushrooms.
Server: That will be C$12.87. What's your address?
Client: I'm at the Delta Victoria, 45 Songhees Road, Ascot room.

```
data PizzaMsg = Toppings [Topping] | Size Size | ···

order :: Chan PizzaMsg → IO ()
order ch = do
  Toppings ts ← readChan ch
  (size, ts')  ← getOrderFromUser ts
  writeChan ch (Size size)
  writeChan ch (Toppings ts')
```

Problem: Ordering a Pizza

Client: Hi. What pizza toppings do you have?
  Server: We have asparagus, broccoli, cauliflower, . . .
Client: I'd like a medium pizza with olives and mushrooms.
  Server: That will be C$12.87. What's your address?
Client: I'm at the Delta Victoria, 45 Songhees Road, Ascot room.

```
data PizzaMsg = Toppings [Topping] | Size Size | ···

order :: Chan PizzaMsg → IO ()
order ch = do
  Toppings ts ← readChan ch
  (size, ts')  ← getOrderFromUser ts
  writeChan ch (Size size)
  writeChan ch (Toppings ts')
```

Problem: Ordering a Pizza

Client: Hi. What pizza toppings do you have?
  Server: We have asparagus, broccoli, cauliflower, . . .
Client: I'd like a medium pizza with olives and mushrooms.
  Server: That will be C$12.87. What's your address?
Client: I'm at the Delta Victoria, 45 Songhees Road, Ascot room.

```
data PizzaMsg = Toppings [Topping] | Size Size | ···

order :: Chan PizzaMsg → IO ()
order ch = do
  Toppings ts ← readChan ch
  (size, ts')   ← getOrderFromUser ts
  writeChan ch (Size size)
  writeChan ch (Toppings ts')
```

Problem: Ordering a Pizza

Client: Hi. What pizza toppings do you have?
  Server: We have asparagus, broccoli, cauliflower, . . .
Client: I'd like a medium pizza with olives and mushrooms.
  Server: That will be C$12.87. What's your address?
Client: I'm at the Delta Victoria, 45 Songhees Road, Ascot room.

```
data PizzaMsg = Toppings [Topping] | Size Size | ···

order :: Chan PizzaMsg → IO ()
order ch = do
  Toppings ts ← readChan ch
  (size, ts')  ← getOrderFromUser ts
  writeChan ch (Toppings ts')
  writeChan ch (Size size)
```

Solution: Session Types

We want to say *ch* is a channel on which we can ...

$ch :: \text{Chan} \cdots$

Solution: Session Types

We want to say *ch* is a channel on which we can

1. receive a list of toppings ...

$ch :: \text{Chan}\,([\text{Topping}]\ ?\ \cdots)$

Solution: Session Types

We want to say *ch* is a channel on which we can

1. receive a list of toppings,
2. send a size . . .

$ch :: \text{Chan}\,([\text{Topping}]\;?\;\text{Size}\;!\;\cdots)$

Solution: Session Types

We want to say *ch* is a channel on which we can

1. receive a list of toppings,
2. send a size,
3. send a list of toppings . . .

ch :: Chan ([Topping] ? Size ! [Topping] ! $\cdots$)

Solution: Session Types

We want to say *ch* is a channel on which we can

1. receive a list of toppings,
2. send a size,
3. send a list of toppings,
4. receive a price ...

ch :: Chan ([Topping] ? Size ! [Topping] ! Price ? $\cdots$)

Solution: Session Types

We want to say *ch* is a channel on which we can

1. receive a list of toppings,
2. send a size,
3. send a list of toppings,
4. receive a price,
5. send an address . . .

$ch :: \text{Chan}\,([\text{Topping}]\ ?\ \text{Size}\ !\ [\text{Topping}]\ !\ \text{Price}\ ?\ \text{Address}\ !\ \cdots)$

Solution: Session Types

We want to say *ch* is a channel on which we can

1. receive a list of toppings,
2. send a size,
3. send a list of toppings,
4. receive a price,
5. send an address, and finally
6. hang up the phone.

ch :: Chan ([Topping] ? Size ! [Topping] ! Price ? Address ! ϵ)

Outline

1. Introduction
   - A Pizza Order Protocol
   - Background
2. The Details
   - Tour of Session Types
   - Implementation: A Single Implicit Channel
   - Live Demonstration
   - Correctness
3. Conclusion
   - Bonus Features
   - Similar Implementations
   - Future Work

A Brief History of Session Types

- Proposed as a type system for the π calculus (Gay & Hole 1999)
- A variety of calculi: π-like, λ-like, object-like

A Brief History of Session Types

- Proposed as a type system for the π calculus (Gay & Hole 1999)
- A variety of calculi: π-like, λ-like, object-like

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon ?D.S; \mathsf{receive}\ v \mapsto \Sigma; D; \alpha\colon S} \quad \text{(C-ReceiveD)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha \qquad c \text{ fresh}}{\Gamma; \Sigma, \alpha\colon ?S'.S; \mathsf{receive}\ v \mapsto \Sigma; \mathsf{Chan}\ c; c\colon S', \alpha\colon S} \quad \text{(C-ReceiveS)}$$

$$\frac{\Gamma; v \mapsto D \qquad \Gamma; v' \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon !D.S; \mathsf{send}\ v\ \mathsf{on}\ v' \mapsto \Sigma; \mathsf{Unit}; \alpha\colon S} \quad \text{(C-SendD)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \beta \qquad \Gamma; v' \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon !S'.S, \beta\colon S'; \mathsf{send}\ v\ \mathsf{on}\ v' \mapsto \Sigma; \mathsf{Unit}; \alpha\colon S} \quad \text{(C-SendS)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha \qquad j \in I}{\Gamma; \Sigma, \alpha\colon \oplus\langle l_i\colon S_i\rangle_{i \in I}; \mathsf{select}\ l_j\ \mathsf{on}\ v \mapsto \Sigma; \mathsf{Unit}; \alpha\colon S_j} \quad \text{(C-Select)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha \qquad \forall j \in I.(\Gamma; \Sigma, \alpha\colon S_j; e_j \mapsto \Sigma_1; T; \Sigma_2)}{\Gamma; \Sigma, \alpha\colon \&\langle l_i\colon S_i\rangle_{i \in I}; \mathsf{case}\ v\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i \in I} \mapsto \Sigma_1; T; \Sigma_2} \quad \text{(C-Case)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon \mathsf{End}; \mathsf{close}\ v \mapsto \Sigma; \mathsf{Unit}; \emptyset} \quad \text{(C-Close)}$$

$$\frac{\Gamma; v \mapsto [S] \qquad c \text{ fresh}}{\Gamma; \Sigma; \mathsf{accept}\ v \mapsto \Sigma; \mathsf{Chan}\ c; c\colon S} \quad \text{(C-Accept)}$$

$$\frac{\Gamma; v \mapsto [S] \qquad c \text{ fresh}}{\Gamma; \Sigma; \mathsf{request}\ v \mapsto \Sigma; \mathsf{Chan}\ c; c\colon \overline{S}} \quad \text{(C-Request)}$$

$$\frac{\Gamma; v \mapsto T}{\Gamma; \Sigma; v \mapsto \Sigma; T; \emptyset} \quad \text{(C-Val)}$$

$$\frac{\Gamma; v \mapsto (\Sigma; T \to U; \Sigma') \qquad \Gamma; v' \mapsto T}{\Gamma; \Sigma, \Sigma''; v v' \mapsto \Sigma''; U; \Sigma'} \quad \text{(C-App)}$$

$$\Gamma; \Sigma; \mathsf{new}\ S \mapsto \Sigma; [S]; \emptyset \quad \text{(C-New)}$$

$$\frac{\Gamma; \Sigma; e \mapsto \Sigma_1; T_1; \Sigma'_1 \qquad \Gamma, x\colon T_1; \Sigma_1, \Sigma'_1; t \mapsto \Sigma_2; T_2; \Sigma'_2}{\Gamma; \Sigma; \mathsf{let}\ x = e\ \mathsf{in}\ t \mapsto \Sigma_1 \cap \Sigma_2; T_2; (\Sigma'_1 \cap \Sigma_2), \Sigma'_2} \quad \text{(C-Let)}$$

$$\frac{\Gamma; \Sigma; t_1 \mapsto \Sigma_1; T_1; \emptyset \qquad \Gamma; \Sigma_1; t_2 \mapsto \Sigma_2; T_2; \emptyset}{\Gamma; \Sigma; (\mathsf{fork}\ t_1; t_2) \mapsto \Sigma_2; T_2; \emptyset} \quad \text{(C-Fork)}$$

A Brief History of Session Types

- Proposed as a type system for the π calculus (Gay & Hole 1999)
- A variety of calculi: π-like, λ-like, object-like

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon ?D.S; \mathsf{receive}\ v \mapsto \Sigma; D; \alpha\colon S} \quad \text{(C-ReceiveD)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha \qquad c \text{ fresh}}{\Gamma; \Sigma, \alpha\colon ?S'.S; \mathsf{receive}\ v \mapsto \Sigma; \mathsf{Chan}\ c; c\colon S', \alpha\colon S} \quad \text{(C-ReceiveS)}$$

$$\frac{\Gamma; v \mapsto D \qquad \Gamma; v' \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon !D.S; \mathsf{send}\ v\ \mathsf{on}\ v' \mapsto \Sigma; \mathsf{Unit}; \alpha\colon S} \quad \text{(C-SendD)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \beta \qquad \Gamma; v' \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon !S'.S, \beta\colon S'; \mathsf{send}\ v\ \mathsf{on}\ v' \mapsto \Sigma; \mathsf{Unit}; \alpha\colon S} \quad \text{(C-SendS)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha \qquad j \in I}{\Gamma; \Sigma, \alpha\colon \oplus\langle l_i\colon S_i\rangle_{i\in I}; \mathsf{select}\ l_j\ \mathsf{on}\ v \mapsto \Sigma; \mathsf{Unit}; \alpha\colon S_j} \quad \text{(C-Select)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha \qquad \forall j \in I.(\Gamma; \Sigma, \alpha\colon S_j; e_j \mapsto \Sigma_1; T; \Sigma_2)}{\Gamma; \Sigma, \alpha\colon \&\langle l_i\colon S_i\rangle_{i\in I}; \mathsf{case}\ v\ \mathsf{of}\ \{l_i \Rightarrow e_i\}_{i\in I} \mapsto \Sigma_1; T; \Sigma_2} \quad \text{(C-Case)}$$

$$\frac{\Gamma; v \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha\colon \mathsf{End}; \mathsf{close}\ v \mapsto \Sigma; \mathsf{Unit}; \emptyset} \quad \text{(C-Close)}$$

$$\frac{\Gamma; v \mapsto [S] \qquad c \text{ fresh}}{\Gamma; \Sigma; \mathsf{accept}\ v \mapsto \Sigma; \mathsf{Chan}\ c; c\colon S} \quad \text{(C-Accept)}$$

$$\frac{\Gamma; v \mapsto [S] \qquad c \text{ fresh}}{\Gamma; \Sigma; \mathsf{request}\ v \mapsto \Sigma; \mathsf{Chan}\ c; c\colon \overline{S}} \quad \text{(C-Request)}$$

$$\frac{\Gamma; v \mapsto T}{\Gamma; \Sigma; v \mapsto \Sigma; T; \emptyset} \quad \text{(C-Val)}$$

$$\frac{\Gamma; v \mapsto (\Sigma; T \to U; \Sigma') \qquad \Gamma; v' \mapsto T}{\Gamma; \Sigma, \Sigma''; vv' \mapsto \Sigma''; U; \Sigma'} \quad \text{(C-App)}$$

$$\Gamma; \Sigma; \mathsf{new}\ S \mapsto \Sigma; [S]; \emptyset \quad \text{(C-New)}$$

$$\frac{\Gamma; \Sigma; e \mapsto \Sigma_1; T_1; \Sigma_1' \qquad \Gamma, x\colon T_1; \Sigma_1, \Sigma_1'; t \mapsto \Sigma_2; T_2; \Sigma_2'}{\Gamma; \Sigma; \mathsf{let}\ x = e\ \mathsf{in}\ t \mapsto \Sigma_1 \cap \Sigma_2; T_2; (\Sigma_1' \cap \Sigma_2), \Sigma_2'} \quad \text{(C-Let)}$$

$$\frac{\Gamma; \Sigma; t_1 \mapsto \Sigma_1; T_1; \emptyset \qquad \Gamma; \Sigma_1; t_2 \mapsto \Sigma_2; T_2; \emptyset}{\Gamma; \Sigma; (\mathsf{fork}\ t_1; t_2) \mapsto \Sigma_2; T_2; \emptyset} \quad \text{(C-Fork)}$$

- How about linear types?

$$\textit{send} :: \forall \alpha, \beta.\, \mathsf{Chan}\,(\alpha\ !\ \beta) \multimap \alpha \multimap \mathsf{Chan}\,\beta$$

This suggested a natural implementation in Haskell.

Our Contributions

Our Haskell session types library:

- works with existing concurrency mechanisms;
- handles multiple communication channels; and
- infers session types automatically.

Syntax and Semantics of Session Types

$s ::=$	$a\,!\,s$	send an *a*, then do *s*
$\mid$	$a\,?\,s$	receive an *a*, then do *s*
$\mid$	ϵ	the empty/finished session

Syntax and Semantics of Session Types

$s ::=$	$a \,!\, s$	send an a, then do s
$\mid$	$a \,?\, s$	receive an a, then do s
$\mid$	ϵ	the empty/finished session
$\mid$	$s_1 \oplus s_2$	internal choice between s_1 and s_2
$\mid$	$s_1 \,\&\, s_2$	external choice between s_1 and s_2

Syntax and Semantics of Session Types

data	a :!: s	— send an a, then do s
data	a :?: s	— receive an a, then do s
data	Eps	— the empty/finished session
data	s_1 :⊕: s_2	— internal choice between s_1 and s_2
data	s_1 :&: s_2	— external choice between s_1 and s_2

Duality

Suppose one process has a channel with session type

$$(\text{Int} :!: \text{Int} :?: \text{Eps}) :\oplus: (\text{Int} :!: \text{String} :!: \text{Int} :?: \text{Eps}).$$

What is the type of the other end of the channel?

Duality

Suppose one process has a channel with session type

$$(\text{Int} :!: \text{Int} :?: \text{Eps}) :\oplus: (\text{Int} :!: \text{String} :!: \text{Int} :?: \text{Eps}).$$

What is the type of the other end of the channel?

$$(\text{Int} \quad \text{Int} \quad \text{Eps}) \quad (\text{Int} \quad \text{String} \quad \text{Int} \quad \text{Eps})$$

Duality

Suppose one process has a channel with session type

$$(\text{Int} :!: \text{Int} :?: \text{Eps}) :\oplus: (\text{Int} :!: \text{String} :!: \text{Int} :?: \text{Eps}).$$

What is the type of the other end of the channel?

$$(\text{Int} :?: \text{Int} :!: \text{Eps}) \qquad (\text{Int} :?: \text{String} :?: \text{Int} :!: \text{Eps})$$

Duality

Suppose one process has a channel with session type

$$(\text{Int} :!: \text{Int} :?: \text{Eps}) :\oplus: (\text{Int} :!: \text{String} :!: \text{Int} :?: \text{Eps}).$$

What is the type of the other end of the channel?

$$(\text{Int} :?: \text{Int} :!: \text{Eps}) :\&: (\text{Int} :?: \text{String} :?: \text{Int} :!: \text{Eps})$$

Duality Inference Rules

Judgment Dual $s_1 \ s_2$

$$\frac{}{\text{Dual Eps Eps}}$$

$$\frac{\text{Dual } s\ s'}{\text{Dual } (a :!: s)\ (a :?: s')} \qquad \frac{\text{Dual } s\ s' \qquad \text{Dual } r\ r'}{\text{Dual } (s :\oplus: r)\ (s' :\&: r')}$$

$$\frac{\text{Dual } s\ s'}{\text{Dual } (a :?: s)\ (a :!: s')} \qquad \frac{\text{Dual } s\ s' \qquad \text{Dual } r\ r'}{\text{Dual } (s :\&: r)\ (s' :\oplus: r')}$$

Duality Inference Rules

$$\textbf{class}\ \text{Dual}\ s_1\ s_2 \mid s_1 \rightsquigarrow s_2, s_2 \rightsquigarrow s_1$$

$$\frac{}{\text{Dual Eps Eps}}$$

$$\frac{\text{Dual}\ s\ s'}{\text{Dual}\ (a :!: s)\ (a :?: s')} \qquad \frac{\text{Dual}\ s\ s' \qquad \text{Dual}\ r\ r'}{\text{Dual}\ (s :\oplus: r)\ (s' :\&: r')}$$

$$\frac{\text{Dual}\ s\ s'}{\text{Dual}\ (a :?: s)\ (a :!: s')} \qquad \frac{\text{Dual}\ s\ s' \qquad \text{Dual}\ r\ r'}{\text{Dual}\ (s :\&: r)\ (s' :\oplus: r')}$$

Duality Inference Rules

$$\textbf{class}\ \mathrm{Dual}\ s_1\ s_2 \mid s_1 \rightsquigarrow s_2, s_2 \rightsquigarrow s_1$$

$$\frac{}{\textbf{instance}\ \mathrm{Dual\ Eps\ Eps}}$$

$$\frac{\textbf{instance}\ \mathrm{Dual}\ s\ s'}{\Rightarrow \mathrm{Dual}\ (a\ {:}!{:}\ s)\ (a\ {:}?{:}\ s')}$$

$$\frac{\textbf{instance}\ (\mathrm{Dual}\ s\ s', \quad \mathrm{Dual}\ r\ r')}{\Rightarrow \mathrm{Dual}\ (s\ {:}\oplus{:}\ r)\ (s'\ {:}\&{:}\ r')}$$

$$\frac{\textbf{instance}\ \mathrm{Dual}\ s\ s'}{\Rightarrow \mathrm{Dual}\ (a\ {:}?{:}\ s)\ (a\ {:}!{:}\ s')}$$

$$\frac{\textbf{instance}\ (\mathrm{Dual}\ s\ s', \quad \mathrm{Dual}\ r\ r')}{\Rightarrow \mathrm{Dual}\ (s\ {:}\&{:}\ r)\ (s'\ {:}\oplus{:}\ r')}$$

Assume Channels

Assume we have untyped synchronous channels:

> **type** UChan
> *unsafeWriteUChan* :: UChan → *a* → IO ()
> *unsafeReadUChan* :: UChan → IO *a*

UChan operations may go wrong (*a la unsafeCoerce#*).

Implementation Problem: Linearity

We've encoded session types, but what about the operations?

$$send :: \mathrm{Chan}\,(a :!: s) \multimap a \multimap IO\,(\mathrm{Chan}\,s)$$

Implementation Problem: Linearity

We've encoded session types, but what about the operations?

$$send :: \mathrm{Chan}\,(a :!: s) \multimap a \multimap IO\,(\mathrm{Chan}\,s)$$

No good!

I claimed a "natural" implementation.

Haskell doesn't have linear types . . .

Implementation Problem: Linearity

We've encoded session types, but what about the operations?

$$send :: \text{Chan}\,(a :!: s) \multimap a \multimap IO\,(\text{Chan}\,s)$$

No good!

I claimed a "natural" implementation.

Haskell doesn't have linear types . . .

but we Haskellers do know how to thread state: a monad.

For session types we must thread not run-time state but compile-time state, so we'll use an *indexed monad*.

An Indexed Monad Class

class IxMonad m **where**

$$(\ggg\!=) :: m\ i\ j\ a \rightarrow (a \rightarrow m\ j\ k\ b) \rightarrow m\ i\ k\ b$$

$$ixret :: a \rightarrow m\ i\ i\ a$$

We expand "**ixdo** notation" to *ixret* and $(\ggg\!=)$ by means of a small preprocessor.

Session Computations

For simplicity,

- one implicit channel, with
- its type maintained by an indexed monad:

newtype Session $s\ s'\ a$

Session Computations

For simplicity,

- one implicit channel, with
- its type maintained by an indexed monad:

newtype Session $s\ s'\ a$ (think: Chan $s \multimap$ (Chan $s' \otimes !a$))

Session Computations

For simplicity,

- one implicit channel, with
- its type maintained by an indexed monad:

newtype Session $s\ s'\ a$ (think: Chan $s \multimap$ (Chan $s' \otimes !a$))

$send\ \ :: a \rightarrow \text{Session}\ (a :!: s)\ s\ ()$

Session Computations

For simplicity,

- one implicit channel, with
- its type maintained by an indexed monad:

newtype Session $s\ s'\ a$ (think: Chan $s \multimap$ (Chan $s' \otimes !a$))

$send$ :: $a \rightarrow$ Session $(a :!: s)\ s\ ()$
(think: $a \rightarrow$ Chan $(a :!: s) \multimap$ (Chan $s' \otimes !()$))

Session Computations

For simplicity,

- one implicit channel, with
- its type maintained by an indexed monad:

$$\textbf{newtype}\ \text{Session}\ s\ s'\ a\ = \text{S}\ \{\ unS :: \text{UChan} \rightarrow \text{IO}\ a\ \}$$

$$send \quad :: a \rightarrow \text{Session}\ (a :!: s)\ s\ ()$$

$$\text{(think: } a \rightarrow \text{Chan}\ (a :!: s) \multimap (\text{Chan}\ s' \otimes !()))$$

Session Computations

For simplicity,

- one implicit channel, with
- its type maintained by an indexed monad:

$$\textbf{newtype}\ \text{Session}\ s\ s'\ a\ = \text{S}\ \{\ \mathit{unS} :: \text{UChan} \rightarrow \text{IO}\ a\ \}$$

$$\begin{aligned}&\mathit{send}\quad :: a \rightarrow \text{Session}\ (a :!: s)\ s\ ()\\ &\mathit{send}\ a = \text{S}\ (\lambda ch \rightarrow \mathit{unsafeWriteUChan}\ ch\ a)\end{aligned}$$

Session Computations

For simplicity,

- one implicit channel, with
- its type maintained by an indexed monad:

$$\textbf{newtype}\ \text{Session}\ s\ s'\ a\ = \text{S}\ \{\ unS :: \text{UChan} \rightarrow \text{IO}\ a\ \}$$

$$\begin{aligned} &send\quad :: a \rightarrow \text{Session}\ (a :!: s)\ s\ () \\ &send\ a = \text{S}\ (\lambda ch \rightarrow unsafeWriteUChan\ ch\ a) \end{aligned}$$

$$\textbf{instance}\ \text{IxMonad Session}\ \textbf{where}\ \ldots$$

The Other Operations

$\rightarrow$ *send* $:: a \rightarrow \text{Session}\ (a :!: s)\ s\ ()$

recv $:: \text{Session}\ (a :?: s)\ s\ a$

close $:: \text{Session Eps}\ ()\ ()$

sel1 $:: \text{Session}\ (s :\oplus: r)\ s\ ()$

sel2 $:: \text{Session}\ (s :\oplus: r)\ r\ ()$

offer $:: \text{Session}\ s\ u\ a \rightarrow \text{Session}\ r\ u\ a \rightarrow \text{Session}\ (s :\&: r)\ u\ a$

The Other Operations

$send :: a \rightarrow \text{Session}\ (a :!: s)\ s\ ()$

$\rightarrow$ $recv :: \text{Session}\ (a :?: s)\ s\ a$

$close :: \text{Session Eps}\ ()\ ()$

$sel1 :: \text{Session}\ (s :\oplus: r)\ s\ ()$

$sel2 :: \text{Session}\ (s :\oplus: r)\ r\ ()$

$offer :: \text{Session}\ s\ u\ a \rightarrow \text{Session}\ r\ u\ a \rightarrow \text{Session}\ (s :\&: r)\ u\ a$

The Other Operations

$$send :: a \rightarrow \text{Session } (a :!: s)\ s\ ()$$

$$recv :: \text{Session } (a :?: s)\ s\ a$$

→ $$close :: \text{Session Eps } ()\ ()$$

$$sel1 :: \text{Session } (s :\oplus: r)\ s\ ()$$

$$sel2 :: \text{Session } (s :\oplus: r)\ r\ ()$$

$$offer :: \text{Session } s\ u\ a \rightarrow \text{Session } r\ u\ a \rightarrow \text{Session } (s :\&: r)\ u\ a$$

The Other Operations

$send \quad :: a \rightarrow \text{Session } (a :!: s) \; s \; ()$

$recv \quad :: \text{Session } (a :?: s) \; s \; a$

$close \quad :: \text{Session Eps } () \; ()$

→ $sel1 \quad :: \text{Session } (s :\oplus: r) \; s \; ()$

$sel2 \quad :: \text{Session } (s :\oplus: r) \; r \; ()$

$offer \quad :: \text{Session } s \; u \; a \rightarrow \text{Session } r \; u \; a \rightarrow \text{Session } (s :\&: r) \; u \; a$

The Other Operations

$$send :: a \to \text{Session}\ (a :!: s)\ s\ ()$$

$$recv :: \text{Session}\ (a :?: s)\ s\ a$$

$$close :: \text{Session Eps}\ ()\ ()$$

$$sel1 :: \text{Session}\ (s :\oplus: r)\ s\ ()$$

→ $$sel2 :: \text{Session}\ (s :\oplus: r)\ r\ ()$$

$$offer :: \text{Session}\ s\ u\ a \to \text{Session}\ r\ u\ a \to \text{Session}\ (s :\&: r)\ u\ a$$

The Other Operations

$send :: a \rightarrow \text{Session}\ (a :!: s)\ s\ ()$

$recv :: \text{Session}\ (a :?: s)\ s\ a$

$close :: \text{Session Eps}\ ()\ ()$

$sel1 :: \text{Session}\ (s :\oplus: r)\ s\ ()$

$sel2 :: \text{Session}\ (s :\oplus: r)\ r\ ()$

→ $offer :: \text{Session}\ s\ u\ a \rightarrow \text{Session}\ r\ u\ a \rightarrow \text{Session}\ (s :\&: r)\ u\ a$

The Other Operations

$$
\begin{array}{ll}
\mathit{send} & ::\ a \rightarrow \text{Session}\ (a :!: s)\ s\ () \\
\rightarrow\ \mathit{send}\ a & =\ \text{S}\ (\lambda ch \rightarrow \mathit{unsafeWriteUChan}\ ch\ a) \\
\mathit{recv} & ::\ \text{Session}\ (a :?: s)\ s\ a \\
\mathit{recv} & =\ \text{S}\ \mathit{unsafeReadUChan} \\
\mathit{close} & ::\ \text{Session Eps}\ ()\ () \\
\mathit{close} & =\ \text{S}\ (\lambda\_ \rightarrow \mathit{return}\ ()) \\
\mathit{sel1} & ::\ \text{Session}\ (s :\oplus: r)\ s\ () \\
\mathit{sel1} & ::\ \text{S}\ (\lambda ch \rightarrow \mathit{unsafeWriteUChan}\ ch\ \text{True}) \\
\mathit{sel2} & ::\ \text{Session}\ (s :\oplus: r)\ r\ () \\
\mathit{sel2} & ::\ \text{S}\ (\lambda ch \rightarrow \mathit{unsafeWriteUChan}\ ch\ \text{False}) \\
\mathit{offer} & ::\ \text{Session}\ s\ u\ a \rightarrow \text{Session}\ r\ u\ a \rightarrow \text{Session}\ (s :\&: r)\ u\ a \\
\mathit{offer}\ s\ r & =\ \text{S}\ (\lambda ch \rightarrow \textbf{do}\ b \leftarrow \mathit{unsafeReadUChan}\ ch \\
 & \qquad\qquad\qquad\quad \textbf{if}\ b\ \textbf{then}\ \mathit{unS}\ s\ ch\ \textbf{else}\ \mathit{unS}\ r\ ch)
\end{array}
$$

The Other Operations

$$
\begin{aligned}
&\mathit{send} && :: a \to \text{Session}\ (a :!: s)\ s\ ()\\
&\mathit{send}\ a && = \text{S}\ (\lambda ch \to \mathit{unsafeWriteUChan}\ ch\ a)\\
&\mathit{recv} && :: \text{Session}\ (a :?: s)\ s\ a\\
\to\ &\mathit{recv} && = \text{S}\ \mathit{unsafeReadUChan}\\
&\mathit{close} && :: \text{Session Eps}\ ()\ ()\\
&\mathit{close} && = \text{S}\ (\lambda\_ \to \mathit{return}\ ())\\
&\mathit{sel1} && :: \text{Session}\ (s :\oplus: r)\ s\ ()\\
&\mathit{sel1} && :: \text{S}\ (\lambda ch \to \mathit{unsafeWriteUChan}\ ch\ \text{True})\\
&\mathit{sel2} && :: \text{Session}\ (s :\oplus: r)\ r\ ()\\
&\mathit{sel2} && :: \text{S}\ (\lambda ch \to \mathit{unsafeWriteUChan}\ ch\ \text{False})\\
&\mathit{offer} && :: \text{Session}\ s\ u\ a \to \text{Session}\ r\ u\ a \to \text{Session}\ (s :\&: r)\ u\ a\\
&\mathit{offer}\ s\ r && = \text{S}\ (\lambda ch \to \textbf{do}\ b \leftarrow \mathit{unsafeReadUChan}\ ch\\
& && \qquad\qquad\qquad \textbf{if}\ b\ \textbf{then}\ \mathit{unS}\ s\ ch\ \textbf{else}\ \mathit{unS}\ r\ ch)
\end{aligned}
$$

The Other Operations

$$
\begin{array}{ll}
\mathit{send} & :: a \rightarrow \mathrm{Session}\ (a :!: s)\ s\ () \\
\mathit{send}\ a & = \mathrm{S}\ (\lambda ch \rightarrow \mathit{unsafeWriteUChan}\ ch\ a) \\
\mathit{recv} & :: \mathrm{Session}\ (a :?: s)\ s\ a \\
\mathit{recv} & = \mathrm{S}\ \mathit{unsafeReadUChan} \\
\mathit{close} & :: \mathrm{Session}\ \mathrm{Eps}\ ()\ () \\
\rightarrow \mathit{close} & = \mathrm{S}\ (\lambda\_ \rightarrow \mathit{return}\ ()) \\
\mathit{sel1} & :: \mathrm{Session}\ (s :\oplus: r)\ s\ () \\
\mathit{sel1} & :: \mathrm{S}\ (\lambda ch \rightarrow \mathit{unsafeWriteUChan}\ ch\ \mathrm{True}) \\
\mathit{sel2} & :: \mathrm{Session}\ (s :\oplus: r)\ r\ () \\
\mathit{sel2} & :: \mathrm{S}\ (\lambda ch \rightarrow \mathit{unsafeWriteUChan}\ ch\ \mathrm{False}) \\
\mathit{offer} & :: \mathrm{Session}\ s\ u\ a \rightarrow \mathrm{Session}\ r\ u\ a \rightarrow \mathrm{Session}\ (s :\&: r)\ u\ a \\
\mathit{offer}\ s\ r & = \mathrm{S}\ (\lambda ch \rightarrow \mathbf{do}\ b \leftarrow \mathit{unsafeReadUChan}\ ch \\
 & \qquad\qquad\qquad\quad \mathbf{if}\ b\ \mathbf{then}\ \mathit{unS}\ s\ ch\ \mathbf{else}\ \mathit{unS}\ r\ ch)
\end{array}
$$

The Other Operations

$$
\begin{array}{lll}
\mathit{send} & :: & a \rightarrow \mathrm{Session}\ (a :!: s)\ s\ () \\
\mathit{send}\ a & = & \mathrm{S}\ (\lambda ch \rightarrow \mathit{unsafeWriteUChan}\ ch\ a) \\
\mathit{recv} & :: & \mathrm{Session}\ (a :?: s)\ s\ a \\
\mathit{recv} & = & \mathrm{S}\ \mathit{unsafeReadUChan} \\
\mathit{close} & :: & \mathrm{Session\ Eps}\ ()\ () \\
\mathit{close} & = & \mathrm{S}\ (\lambda\_ \rightarrow \mathit{return}\ ()) \\
\mathit{sel1} & :: & \mathrm{Session}\ (s :\oplus: r)\ s\ () \\
\rightarrow \mathit{sel1} & :: & \mathrm{S}\ (\lambda ch \rightarrow \mathit{unsafeWriteUChan}\ ch\ \mathrm{True}) \\
\mathit{sel2} & :: & \mathrm{Session}\ (s :\oplus: r)\ r\ () \\
\mathit{sel2} & :: & \mathrm{S}\ (\lambda ch \rightarrow \mathit{unsafeWriteUChan}\ ch\ \mathrm{False}) \\
\mathit{offer} & :: & \mathrm{Session}\ s\ u\ a \rightarrow \mathrm{Session}\ r\ u\ a \rightarrow \mathrm{Session}\ (s :\&: r)\ u\ a \\
\mathit{offer}\ s\ r & = & \mathrm{S}\ (\lambda ch \rightarrow \mathbf{do}\ b \leftarrow \mathit{unsafeReadUChan}\ ch \\
 & & \qquad\qquad\qquad \mathbf{if}\ b\ \mathbf{then}\ \mathit{unS}\ s\ ch\ \mathbf{else}\ \mathit{unS}\ r\ ch)
\end{array}
$$

The Other Operations

```
send      :: a → Session (a :!: s) s ()
send a    = S (λch → unsafeWriteUChan ch a)

recv      :: Session (a :?: s) s a
recv      = S unsafeReadUChan

close     :: Session Eps () ()
close     = S (λ_ → return ())

sel1      :: Session (s :⊕: r) s ()
sel1      :: S (λch → unsafeWriteUChan ch True)

sel2      :: Session (s :⊕: r) r ()
→ sel2    :: S (λch → unsafeWriteUChan ch False)

offer     :: Session s u a → Session r u a → Session (s :&: r) u a
offer s r = S (λch → do b ← unsafeReadUChan ch
                        if b then unS s ch else unS r ch)
```

The Other Operations

$$
\begin{array}{ll}
\textit{send} & :: a \rightarrow \text{Session } (a :!: s)\ s\ () \\
\textit{send } a & = \text{S } (\lambda ch \rightarrow \textit{unsafeWriteUChan ch a}) \\
\textit{recv} & :: \text{Session } (a :?: s)\ s\ a \\
\textit{recv} & = \text{S } \textit{unsafeReadUChan} \\
\textit{close} & :: \text{Session Eps } ()\ () \\
\textit{close} & = \text{S } (\lambda\_ \rightarrow \textit{return } ()) \\
\textit{sel1} & :: \text{Session } (s :\oplus: r)\ s\ () \\
\textit{sel1} & :: \text{S } (\lambda ch \rightarrow \textit{unsafeWriteUChan ch } \text{True}) \\
\textit{sel2} & :: \text{Session } (s :\oplus: r)\ r\ () \\
\textit{sel2} & :: \text{S } (\lambda ch \rightarrow \textit{unsafeWriteUChan ch } \text{False}) \\
\textit{offer} & :: \text{Session } s\ u\ a \rightarrow \text{Session } r\ u\ a \rightarrow \text{Session } (s :\&: r)\ u\ a \\
\textit{offer } s\ r & = \text{S } (\lambda ch \rightarrow \textbf{do } b \leftarrow \textit{unsafeReadUChan ch} \\
 & \qquad\qquad\qquad\qquad \textbf{if } b \textbf{ then } \textit{unS s ch } \textbf{else } \textit{unS r ch})
\end{array}
$$

Putting Things Together

Finally, we need a way to run Session computations.

→ **newtype** Rendezvous *s*

$$
\begin{aligned}
&\mathit{newRendezvous} :: \mathrm{IO}\ (\mathrm{Rendezvous}\ s)\\
&\mathit{accept} :: \mathrm{Rendezvous}\ s \rightarrow \mathrm{Session}\ s\ ()\ a \rightarrow \mathrm{IO}\ a\\
&\mathit{request} :: \mathrm{Dual}\ s\ s' \Rightarrow\\
&\qquad \mathrm{Rendezvous}\ s \rightarrow \mathrm{Session}\ s'\ ()\ a \rightarrow \mathrm{IO}\ a
\end{aligned}
$$

Putting Things Together

Finally, we need a way to run Session computations.

$$
\begin{array}{ll}
\textbf{newtype}\ \text{Rendezvous}\ s & \\
\rightarrow \textit{newRendezvous} & :: \text{IO}\ (\text{Rendezvous}\ s) \\
\textit{accept} & :: \text{Rendezvous}\ s \rightarrow \text{Session}\ s\ ()\ a \rightarrow \text{IO}\ a \\
\textit{request} & :: \text{Dual}\ s\ s' \Rightarrow \\
 & \quad \text{Rendezvous}\ s \rightarrow \text{Session}\ s'\ ()\ a \rightarrow \text{IO}\ a
\end{array}
$$

Putting Things Together

Finally, we need a way to run Session computations.

newtype Rendezvous s
newRendezvous :: IO (Rendezvous s)
→ *accept* :: Rendezvous $s \rightarrow$ Session s () $a \rightarrow$ IO a
request :: Dual s $s' \Rightarrow$
Rendezvous $s \rightarrow$ Session s' () $a \rightarrow$ IO a

Putting Things Together

Finally, we need a way to run Session computations.

newtype Rendezvous s
$newRendezvous$ :: IO (Rendezvous s)
$accept$ :: Rendezvous $s \to$ Session s () $a \to$ IO a
→ $request$ :: Dual $s\ s' \Rightarrow$
Rendezvous $s \to$ Session s' () $a \to$ IO a

Putting Things Together

Finally, we need a way to run Session computations.

$$
\begin{array}{ll}
\textbf{newtype}\ \text{Rendezvous}\ s & \\
\mathit{newRendezvous} & :: \text{IO}\ (\text{Rendezvous}\ s) \\
\mathit{accept} & :: \text{Rendezvous}\ s \rightarrow \text{Session}\ s\ ()\ a \rightarrow \text{IO}\ a \\
\mathit{request} & :: \text{Dual}\ s\ s' \Rightarrow \\
& \quad \text{Rendezvous}\ s \rightarrow \text{Session}\ s'\ ()\ a \rightarrow \text{IO}\ a
\end{array}
$$

→

$$
\begin{array}{ll}
\mathit{connect} & :: \text{Dual}\ s\ s' \Rightarrow \\
& \quad \text{Session}\ s\ ()\ () \rightarrow \text{Session}\ s'\ ()\ a \rightarrow \text{IO}\ a \\
\mathit{connect\ server\ client} & \\
& = \textbf{do}\ \mathit{rv} \leftarrow \mathit{newRendezvous} \\
& \qquad \mathit{forkIO}\ (\mathit{accept\ rv\ server}) \\
& \qquad \mathit{request\ rv\ client}
\end{array}
$$

Putting Things Together

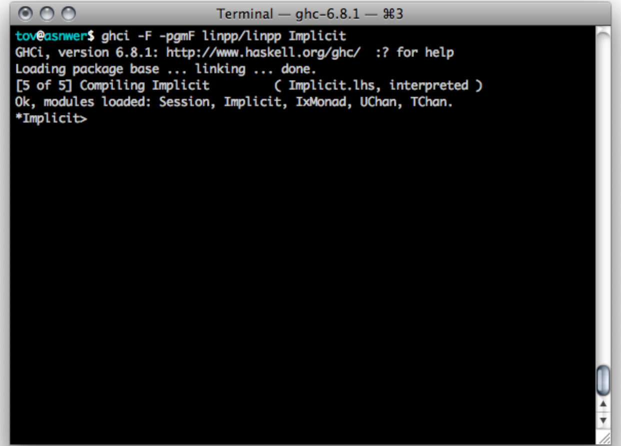
Finally, we need a way to run Session computations.

newtype Rendezvous s
$newRendezvous$ :: IO (Rendezvous s)
$accept$ :: Rendezvous $s \rightarrow$ Session s () $a \rightarrow$ IO a
$request$ :: Dual s $s' \Rightarrow$
Rendezvous $s \rightarrow$ Session s' () $a \rightarrow$ IO a

$connect$ :: Dual s $s' \Rightarrow$
Session s () () $\rightarrow$ Session s' () $a \rightarrow$ IO a

→ $connect\ server\ client$
= **do** $rv \leftarrow newRendezvous$
$forkIO\ (accept\ rv\ server)$
$request\ rv\ client$

Live Demonstration



Correctness

Our library uses unsafe channel operations. Why should we believe it implements session types correctly?

Correctness

Our library uses unsafe channel operations. Why should we believe it implements session types correctly?

We define two calculi:

- $\lambda^{F||F}$ has unsafe channels (like UChan)
- $\lambda_\ell^{F||F}$ has session-typed channels
- $\mathscr{L}[\![\cdot]\!] : \lambda_\ell^{F||F} \to \lambda^{F||F}$ replaces channel operations with the library definitions

Correctness

Our library uses unsafe channel operations. Why should we believe it implements session types correctly?

We define two calculi:

- $\lambda^{F||F}$ has unsafe channels (like UChan)
- $\lambda_\ell^{F||F}$ has session-typed channels
- $\mathscr{L}[\![\cdot]\!] : \lambda_\ell^{F||F} \to \lambda^{F||F}$ replaces channel operations with the library definitions

Theorem (Library Soundness)

If $\vdash_\ell p : \pi$ in $\lambda_\ell^{F||F}$, then in $\lambda^{F||F}$ either

- *$\mathscr{L}[\![p]\!]$ diverges or*
- *$\mathscr{L}[\![p]\!] \Longrightarrow^* w$ where $\vdash w : \mathscr{L}[\![\pi]\!]$.*

Recursion

Extend the syntax of session types:

$$
\begin{array}{rll}
s ::= & \mu v.\, s & \text{recursive session type} \\
| & v & \text{variable instance}
\end{array}
$$

Recursion

Extend the syntax of session types:

data	Rec *s*	— recursive session type
data	Var *n*	— de Bruijn index

Recursion

Extend the syntax of session types:

$$\begin{array}{lll} \textbf{data} & \mathrm{Rec}\ s & \text{— recursive session type} \\ \textbf{data} & \mathrm{Var}\ n & \text{— de Bruijn index} \end{array}$$

And add Dual instances:

$$\frac{}{\mathrm{Dual}\ (\mathrm{Var}\ n)\ (\mathrm{Var}\ n)} \qquad \frac{\mathrm{Dual}\ s\ s'}{\mathrm{Dual}\ (\mathrm{Rec}\ s)\ (\mathrm{Rec}\ s')}$$

Recursion

Extend the syntax of session types:

$$
\begin{array}{lll}
\textbf{data} & \mathsf{Rec}\ s & \text{— recursive session type} \\
\textbf{data} & \mathsf{Var}\ n & \text{— de Bruijn index}
\end{array}
$$

And add Dual instances:

$$
\frac{}{\textbf{instance}\ \mathsf{Dual}\ (\mathsf{Var}\ n)\ (\mathsf{Var}\ n)}
\qquad
\frac{\textbf{instance}\ \mathsf{Dual}\ s\ s'}{\Rightarrow \mathsf{Dual}\ (\mathsf{Rec}\ s)\ (\mathsf{Rec}\ s')}
$$

Recursion

Extend the syntax of session types:

$$\begin{array}{lll} \textbf{data} & \text{Rec}\, s & \text{— recursive session type} \\ \textbf{data} & \text{Var}\, n & \text{— de Bruijn index} \end{array}$$

And add Dual instances:

$$\frac{}{\textbf{instance}\ \text{Dual}\ (\text{Var}\ n)\ (\text{Var}\ n)} \qquad \frac{\textbf{instance}\ \text{Dual}\ s\ s'}{\Rightarrow \text{Dual}\ (\text{Rec}\ s)\ (\text{Rec}\ s')}$$

Each session type must be closed in an environment e, in which we maintain a stack of the bodies of each enclosing Rec:

$$send :: a \rightarrow \text{Session}\ (e, a :!: s)\ (e, s)\ ()$$

Multiple Channels

To keep track of multiple, independent channels:

- Replace Session's single session type with a stack of session types.
- Operations act on the top of the stack.

  $send$ :: Session (a :!: s, x) (s, x) ()
- Provide simple stack shuffling operations.

Multiple Channels

To keep track of multiple, independent channels:

- Replace Session's single session type with a stack of session types.
- Operations act on the top of the stack.

  $send$ :: Session (a :!: s, x) (s, x) ()
- Provide simple stack shuffling operations.

Name-based access would be nice. We can do it with type classes, but it doesn't play well with inference or . . .

Other Languages (Have No Class)

What features might be difficult?

Other Languages (Have No Class)

What features might be difficult?

- Duality
  - The type class translates directly into ML functors
  - In other languages value-level proofs are possible—requires abstraction to avoid witness forgery

Other Languages (Have No Class)

What features might be difficult?

- Duality
  - The type class translates directly into ML functors
  - In other languages value-level proofs are possible—requires abstraction to avoid witness forgery
- The indexed monad requires some sort of higher-orderness and polymorphism

Other Languages (Have No Class)

What features might be difficult?

- Duality
  - The type class translates directly into ML functors
  - In other languages value-level proofs are possible—requires abstraction to avoid witness forgery
- The indexed monad requires some sort of higher-orderness and polymorphism

We have working prototypes in SML, OCaml, Java 1.5, Scala, and C#.

Other Haskell Implementations

- Neubauer and Thiemann. An implementation of session types (PADL'04)

    Single processes speaking wire protocols

- Sackman and Eisenbach. Session types in Haskell: Updating message passing for the 21st century (2008)

    Very full featured; heavy-duty type class hackage

Other Haskell Implementations

- Neubauer and Thiemann. An implementation of session types (PADL'04)

  Single processes speaking wire protocols

- Sackman and Eisenbach. Session types in Haskell: Updating message passing for the 21st century (2008)

  Very full featured; heavy-duty type class hackage

	N&T	**S&E**	**this work**

Other Haskell Implementations

- Neubauer and Thiemann. An implementation of session types (PADL'04)

  Single processes speaking wire protocols

- Sackman and Eisenbach. Session types in Haskell: Updating message passing for the 21st century (2008)

  Very full featured; heavy-duty type class hackage

	N&T	**S&E**	**this work**
branching	binary	*k*-ary	binary

Other Haskell Implementations

- Neubauer and Thiemann. An implementation of session types (PADL'04)

  Single processes speaking wire protocols

- Sackman and Eisenbach. Session types in Haskell: Updating message passing for the 21st century (2008)

  Very full featured; heavy-duty type class hackage

	N&T	S&E	this work
branching	binary	*k*-ary	binary
recursion	named	named	numbered

Other Haskell Implementations

- Neubauer and Thiemann. An implementation of session types (PADL'04)

  Single processes speaking wire protocols

- Sackman and Eisenbach. Session types in Haskell: Updating message passing for the 21st century (2008)

  Very full featured; heavy-duty type class hackage

	N&T	**S&E**	**this work**
branching	binary	k-ary	binary
recursion	named	named	numbered
multiple channels	no	named	numbered

Other Haskell Implementations

- Neubauer and Thiemann. An implementation of session types (PADL'04)

  Single processes speaking wire protocols

- Sackman and Eisenbach. Session types in Haskell: Updating message passing for the 21st century (2008)

  Very full featured; heavy-duty type class hackage

	N&T	S&E	this work
branching	binary	*k*-ary	binary
recursion	named	named	numbered
multiple channels	no	named	numbered
type inference	N/A	no	yes

Other Haskell Implementations

- Neubauer and Thiemann. An implementation of session types (PADL'04)

  Single processes speaking wire protocols
- Sackman and Eisenbach. Session types in Haskell: Updating message passing for the 21st century (2008)

  Very full featured; heavy-duty type class hackage

	N&T	**S&E**	**this work**
branching	binary	*k*-ary	binary
recursion	named	named	numbered
multiple channels	no	named	numbered
type inference	N/A	no	yes
soundness theorem	sort of	partial	yes

A Problem, an Opportunity

How should exceptions work?

- We can throw (if channels are affine)
- But we can't catch within a session
- Would it be profitable to combine with STM?

Thank You

Contact us:

- tov@ccs.neu.edu
- http://www.ccs.neu.edu/~tov/session-types/