

Introduction to Session Types

Lecture 1: Binary Session Types

Simon Fowler

University of Glasgow

SPLV 2023

Hello!

- Lecturer in PL at the University of Glasgow
- (Did my BSc at St Andrews, so very happy to be back!)
- Interested in behavioural types, multi-tier programming
- Especially interested in incorporating behavioural types (including session types) in language designs



Lecture 1: Binary Session Types

- What **are** session types, anyway?

- A session-typed process calculus

- A session-typed λ -calculus

- Implementation: Session Types in Links

Lecture 2: Multiparty Session Types and Tools

- Global and local types

- A session-typed multiparty process calculus

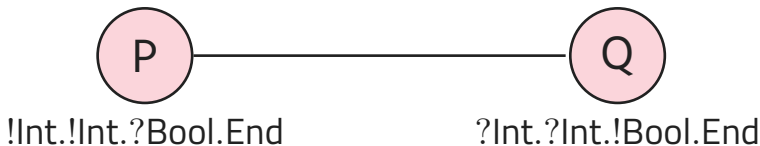
- Implementations and extensions

Communication-centric programming languages



Communication-centric programming languages support concurrency using **explicit message passing**, rather than sharing memory.

Explicit message passing is not a silver bullet!




Session: Structured interaction between two participants

Session type: Type of a **channel endpoint**

Allows conformance to a protocol to be checked before the program is run

Bank of Sessions




Bank of Sessions

User

Sign in



Bank of Sessions



Bank of Sessions

It seems you are logging on from a new device.


To ensure security of your account, please enter the following key into your hardware token, and enter the digits shown on the screen.

Key: 67539

Submit



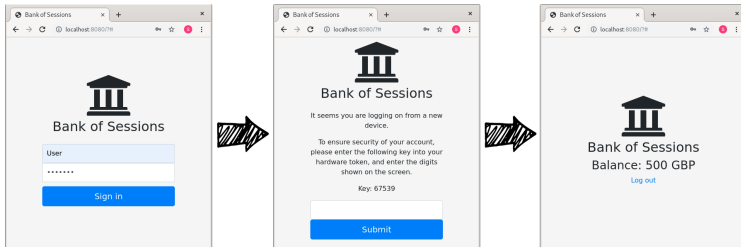
Bank of Sessions



Bank of Sessions

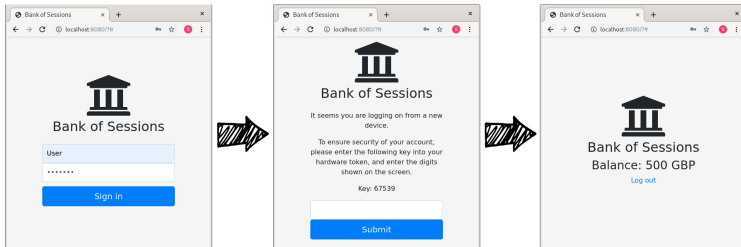
Balance: 500 GBP

[Log out](#)



TwoFactorClient \triangleq

```
!(Username, Password).&{  
  Authenticated : ClientBody,  
  Challenge : ?ChallengeKey.!Response.  
    &{Authenticated : ClientBody,  
      AccessDenied : end},  
  AccessDenied : end  
}
```



TwoFactorClient \triangleq

```
!(Username, Password).&{
  Authenticated : ClientBody,
  Challenge : ?ChallengeKey.!Response.
    &{Authenticated : ClientBody,
      AccessDenied : end},
  AccessDenied : end
}
```

TwoFactorServer \triangleq

```
?(Username, Password). $\oplus$ {
  Authenticated : ServerBody,
  Challenge : !ChallengeKey.?Response.
     $\oplus$ {Authenticated : ServerBody,
      AccessDenied : end},
  AccessDenied : end
}
```


Two-Factor Authentication

```
twoFactorServer : TwoFactorServer  $\rightarrow$  ()  
twoFactorServer(s)  $\triangleq$   
  let ((username, password), s) = receive s in  
  if checkDetails(username, password) then  
    let s = select Authenticated s in serverMain(s)  
  else  
    let s = select AccessDenied s in close s
```

Correct implementation

Two-Factor Authentication

$s : ?(\text{Username}, \text{Password}). \oplus \{ \dots \}$

Authenticated : ServerBody;
 $s : \oplus \{ \text{Challenge} : \dots ;$
AccessDenied : End }

twoFactorServer : TwoFactorServer $\rightarrow ()$
 $\text{twoFactorServer}(s) \triangleq$

$s : \text{ServerBody}$

let ((username, password), s) = **receive** s **in**

if checkDetails(username, password) **then**

let s = **select** Authenticated s **in** serverMain(s)

else

let s = **select** AccessDenied s **in close** s

$s : \text{End}$

Correct implementation

Two-Factor Authentication

```
twoFactorServer : TwoFactorServer  $\multimap$  ()  
twoFactorServer(s)  $\triangleq$   
  let s = send "Hello" s in  
    let ((username, password), s) = receive s in  
      if checkDetails(username, password) then  
        let s = select Authenticated s in serverMain(s)  
      else  
        let s = select AccessDenied s in close s
```

Incorrect: does not follow protocol

Two-Factor Authentication

```
twoFactorServer : TwoFactorServer  $\multimap$  ()  
twoFactorServer(s)  $\triangleq$   
  let ((username, password), s) = receive s in  
  ()
```

Incorrect: does not implement entire protocol

Two-Factor Authentication

```
twoFactorServer : TwoFactorServer  $\multimap$  ()  
twoFactorServer(s)  $\triangleq$   
  let ((username, password), t) = receive s in  
  let ((username, password), s) = receive s in  
  if checkDetails(username, password) then  
    let s = select Authenticated s in serverMain(s)  
  else  
    let s = select AccessDenied s in close s
```

Incorrect: uses endpoint twice

Requires **linearity**

Properties of Session Types

Session Fidelity

Communication follows the session type

Deadlock-Freedom

Communication does not get stuck

Usually within a **single session**, but we will see how to get global progress later

A Bit of Context

This year marks **30 years** of session types!

Original 3 papers:

- K. Honda. Types for Dyadic Interaction. CONCUR 1993.
- Takekuchi, Honda, and Kubo. An Interaction-Based Language and its Type System. PARLE 1994.
- Honda, Vasconcelos, and Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. ESOP 1998.

Session types often crop up at major PL conferences (POPL, ICFP, ECOOP, ESOP...)

A Session-Typed Process Calculus

Process Calculi?

Process calculi model **concurrent computation**

Much simpler structure than a programming language

Early work on session types was, and much of today's work still is, in setting of process calculi

We will start with a process calculus, and then build up to a λ -calculus

Session Process Calculus

Types

$A, B ::= \text{Int} \mid \text{Bool} \mid S$

Session types

| | | |
|------------|------------------------------------|--|
| $S, T ::=$ | $!A.S$ | Send value of type A , continue as S |
| | $ \quad ?A.S$ | Receive value of type A , continue as S |
| | $ \quad \oplus\{\ell_i . S_i\}_i$ | Select an ℓ_i , continue as S_i |
| | $ \quad \&\{\ell_i . S_i\}_i$ | Offer branches ℓ_i with continuations S_i |
| | $ \quad \text{end}$ | Finished |

Loosely based on Vasconcelos, V. T. (2012). **Fundamentals of session types**. Inf. Comput. 217, 52-70.

Session Process Calculus

Values

Values $V, W ::= x \mid n \mid \text{true} \mid \text{false} \mid \dots$

Processes

| | | |
|---------------|---|---|
| $P, Q, R ::=$ | 0 | Inactive process |
| | $(\nu xy)P$ | Bind co-names x and y in P |
| | $P \parallel Q$ | Parallel composition |
| | $x[V].P$ | Send V along x , continue as P |
| | $x(z).P$ | Receive along x , binding result to z ; continue as P |
| | $x \triangleleft \ell.P$ | Select label ℓ on x ; continue as P |
| | $x \triangleright \{\ell_i . P_i\}_{i \in I}$ | Offer on x ; each label ℓ_i has continuation P_i |

Type system: Main ideas

Linearity: Each non-finished session-typed name must be used precisely once at the top-level (re-bound by communication)

Duality: Types of co-names are dual, so where one sends, the other receives

$$\overline{!Int.!Int.?Int.end} = ?Int.?Int.!Int.end$$

Type system: Main ideas

Linearity: Each non-finished session-typed name must be used precisely once at the top-level (re-bound by communication)

Duality: Types of co-names are dual, so where one sends, the other receives

$$\overline{!Int.!Int.?Int.end} = ?Int.?Int.!Int.end$$

$$\overline{!A.S} = ?A.\bar{S} \qquad \overline{?A.S} = !A.\bar{S} \qquad \overline{\oplus\{\ell_i . S_i\}_{i \in I}} = \&\{\ell_i . \bar{S}_i\}_{i \in I}$$

$$\overline{\&\{\ell_i . S_i\}_{i \in I}} = \oplus\{\ell_i . \bar{S}_i\}_{i \in I}$$

Typing rules (Values)

$$\frac{\text{un}(\Gamma)}{\Gamma, x : A \vdash x : A} \quad \frac{\text{un}(\Gamma) \quad b \in \{\text{true}, \text{false}\}}{\Gamma \vdash b : \text{Bool}} \quad \frac{\text{un}(\Gamma) \quad n \in \mathbb{Z}}{\Gamma \vdash n : \text{Int}}$$

$\text{un}(\Gamma)$ holds if all types in Γ are either Int , Bool , or finished session type end

$\Gamma_1 + \Gamma_2$ defined if environments Γ_1 and Γ_2 overlap only variables with unrestricted types

Typing rules (Processes, 1)

$$\frac{\text{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}$$

Inactive process typable under unrestricted environment

Typing rules (Processes, 1)

$$\frac{\text{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}$$

Inactive process typable under unrestricted environment

$$\frac{\Gamma, x : S, y : \bar{S} \vdash P}{\Gamma \vdash (\nu xy)P}$$

Name restriction: bind dual endpoints x, y in P

Typing rules (Processes, 1)

$$\frac{\text{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}$$

Inactive process typable under unrestricted environment

$$\frac{\Gamma, x : S, y : \bar{S} \vdash P}{\Gamma \vdash (\nu xy)P}$$

Name restriction: bind dual endpoints x, y in P

$$\frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \parallel Q}$$

Parallel composition: P and Q typable under disjoint environments

Typing rules (Processes, 1)

$$\frac{\text{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}$$

Inactive process typable under unrestricted environment

$$\frac{\Gamma, x : S, y : \bar{S} \vdash P}{\Gamma \vdash (\nu xy)P}$$

Name restriction: bind dual endpoints x, y in P

$$\frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \parallel Q}$$

Parallel composition: P and Q typable under disjoint environments

$$\frac{\Gamma_1 \vdash V : A \quad \Gamma_2, x : S \vdash P}{\Gamma_1 + \Gamma_2, x : !A.S \vdash x[V].P}$$

Send: type of V should match session type; bind x to S in continuation

Typing rules (Processes, 2)

$$\frac{\Gamma, x : S, y : A \vdash P}{\Gamma, x : ?A.S \vdash x(y).P}$$

Receive: given channel x of type $?A.S$, bind x to S and y to A in continuation

Typing rules (Processes, 2)

$$\frac{\Gamma, x : S, y : A \vdash P}{\Gamma, x : ?A.S \vdash x(y).P}$$

Receive: given channel x of type $?A.S$, bind x to S and y to A in continuation

$$\frac{j \in I \quad \Gamma, x : S_j \vdash P}{\Gamma, x : \oplus\{\ell_i . S_i\}_{i \in I} \vdash x \triangleleft \ell_j . P}$$

Select: given label j in type, bind x to S_j in continuation

Typing rules (Processes, 2)

$$\frac{\Gamma, x : S, y : A \vdash P}{\Gamma, x : ?A.S \vdash x(y).P}$$

Receive: given channel x of type $?A.S$, bind x to S and y to A in continuation

$$\frac{j \in I \quad \Gamma, x : S_j \vdash P}{\Gamma, x : \oplus \{\ell_i . S_i\}_{i \in I} \vdash x \triangleleft \ell_j . P}$$

Select: given label j in type, bind x to S_j in continuation

$$\frac{(\Gamma, x : S_i \vdash P_i)_{i \in I}}{\Gamma, x : \& \{\ell_i . S_i\}_{i \in I} \vdash x \triangleright \{\ell_i . P_i\}_{i \in I}}$$

Offer: bind x to S_i in each continuation P_i

Well typed?

$(\nu xy)(x[5].\mathbf{0} \parallel y(z).\mathbf{0})$

Well typed?

$(\nu xy)(x[5].\mathbf{0} \parallel y(z).\mathbf{0})$



Well typed?

$(\nu xy)(x[5].\mathbf{0} \parallel y(z).\mathbf{0})$



$(\nu xy)(x[5].\mathbf{0} \parallel \mathbf{0})$

Well typed?

$(\nu xy)(x[5].\mathbf{0} \parallel y(z).\mathbf{0})$



$(\nu xy)(x[5].\mathbf{0} \parallel \mathbf{0})$



Well typed?

$(\nu xy)(x[5].\mathbf{0} \parallel y(z).\mathbf{0})$



$(\nu xy)(x[5].\mathbf{0} \parallel \mathbf{0})$



$(\nu xy)(x[5].\mathbf{0} \parallel x[10].\mathbf{0} \parallel y(z).\mathbf{0})$

Well typed?

$(\nu xy)(x[5].\mathbf{0} \parallel y(z).\mathbf{0})$



$(\nu xy)(x[5].\mathbf{0} \parallel \mathbf{0})$



$(\nu xy)(x[5].\mathbf{0} \parallel x[10].\mathbf{0} \parallel y(z).\mathbf{0})$



Well typed?

$(\nu xy)(x[5].0 \parallel y(z).0)$



$(\nu xy)(x[5].0 \parallel 0)$



$(\nu xy)(x[5].0 \parallel x[10].0 \parallel y(z).0)$



$(\nu xy)(\nu ab)(x[a].0 \parallel y(z).0)$

Well typed?

$(\nu xy)(x[5].\mathbf{0} \parallel y(z).\mathbf{0})$



$(\nu xy)(x[5].\mathbf{0} \parallel \mathbf{0})$



$(\nu xy)(x[5].\mathbf{0} \parallel x[10].\mathbf{0} \parallel y(z).\mathbf{0})$



$(\nu xy)(\nu ab)(x[a].\mathbf{0} \parallel y(z).\mathbf{0})$



Well typed?

$(\nu xy)(x[5].0 \parallel y(z).0)$



$(\nu xy)(x[5].0 \parallel 0)$



$(\nu xy)(x[5].0 \parallel x[10].0 \parallel y(z).0)$



$(\nu xy)(\nu ab)(x[a].0 \parallel y(z).0)$



$(\nu xy)(\nu ab)(x[a].0 \parallel b[5].0 \parallel y(z).0)$

Well typed?

$(\nu xy)(x[5].0 \parallel y(z).0)$



$(\nu xy)(x[5].0 \parallel 0)$



$(\nu xy)(x[5].0 \parallel x[10].0 \parallel y(z).0)$



$(\nu xy)(\nu ab)(x[a].0 \parallel y(z).0)$



$(\nu xy)(\nu ab)(x[a].0 \parallel b[5].0 \parallel y(z).0)$



Well typed?

$(\nu xy)(x[5].0 \parallel y(z).0)$



$(\nu xy)(x[5].0 \parallel 0)$



$(\nu xy)(x[5].0 \parallel x[10].0 \parallel y(z).0)$



$(\nu xy)(\nu ab)(x[a].0 \parallel y(z).0)$



$(\nu xy)(\nu ab)(x[a].0 \parallel b[5].0 \parallel y(z).0)$



$(\nu xy)(x[5].y(z).0)$

Well typed?

$(\nu xy)(x[5].0 \parallel y(z).0)$ ✓

$(\nu xy)(x[5].0 \parallel 0)$ ✗

$(\nu xy)(x[5].0 \parallel x[10].0 \parallel y(z).0)$ ✗

$(\nu xy)(\nu ab)(x[a].0 \parallel y(z).0)$ ✓

$(\nu xy)(\nu ab)(x[a].0 \parallel b[5].0 \parallel y(z).0)$ ✗

$(\nu xy)(x[5].y(z).0)$ ✓

Structural congruence

Does it make sense to distinguish the processes:

$$(\nu xy)(x \triangleleft \ell_1.P \parallel y \triangleright \{\ell_1 . Q_1; \ell_2 . Q_2\}) \qquad (\nu xy)(y \triangleright \{\ell_1 . Q_1; \ell_2 . Q_2\} \parallel x \triangleleft \ell_1.P)$$

Structural congruence

Does it make sense to distinguish the processes:

$$(\nu xy)(x \triangleleft \ell_1.P \parallel y \triangleright \{\ell_1 . Q_1; \ell_2 . Q_2\}) \qquad (\nu xy)(y \triangleright \{\ell_1 . Q_1; \ell_2 . Q_2\} \parallel x \triangleleft \ell_1.P)$$

- **No:** they mean the same thing, and we would have to write many more reduction rules
- Other ways processes can differ in syntax: associativity, scoping, inactive processes...

Structural congruence

Commutativity: $P \parallel Q \equiv Q \parallel P$

Structural congruence

Commutativity: $P \parallel Q \equiv Q \parallel P$

Associativity: $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$

Structural congruence

Commutativity: $P \parallel Q \equiv Q \parallel P$

Associativity: $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$

Identity: $P \parallel 0 \equiv P$

Structural congruence

Commutativity: $P \parallel Q \equiv Q \parallel P$

Associativity: $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$

Identity: $P \parallel 0 \equiv P$

Scope Extrusion: $(\nu xy)(P \parallel Q) \equiv P \parallel (\nu xy)Q$ if $x, y \notin \text{fv}(P)$

Structural congruence

Commutativity: $P \parallel Q \equiv Q \parallel P$

Associativity: $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$

Identity: $P \parallel 0 \equiv P$

Scope Extrusion: $(\nu xy)(P \parallel Q) \equiv P \parallel (\nu xy)Q$ if $x, y \notin \text{fv}(P)$

Ordering within restriction: $(\nu xy)P \equiv (\nu yx)P$

Structural congruence

Commutativity: $P \parallel Q \equiv Q \parallel P$

Associativity: $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$

Identity: $P \parallel 0 \equiv P$

Scope Extrusion: $(\nu xy)(P \parallel Q) \equiv P \parallel (\nu xy)Q$ if $x, y \notin \text{fv}(P)$

Ordering within restriction: $(\nu xy)P \equiv (\nu yx)P$

Ordering of restrictions: $(\nu xx')(\nu yy')P \equiv (\nu yy')(\nu xx')P$

Structural congruence

Commutativity: $P \parallel Q \equiv Q \parallel P$

Associativity: $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$

Identity: $P \parallel 0 \equiv P$

Scope Extrusion: $(\nu xy)(P \parallel Q) \equiv P \parallel (\nu xy)Q$ if $x, y \notin \text{fv}(P)$

Ordering within restriction: $(\nu xy)P \equiv (\nu yx)P$

Ordering of restrictions: $(\nu xx')(\nu yy')P \equiv (\nu yy')(\nu xx')P$

Garbage collection: $(\nu xy)P \equiv P$ if $x, y \notin \text{fv}(P)$

Reduction example

$$(\nu xy)(x[5] . x(a) . \mathbf{0} \parallel y(b) . y[b] . \mathbf{0})$$

Reduction example

$$\begin{array}{c} (\nu xy)(x[5] . x(a) . \mathbf{0} \parallel y(b) . y[b] . \mathbf{0}) \\ \downarrow \\ (\nu xy)(x(a) . \mathbf{0} \parallel y[5] . \mathbf{0}) \end{array}$$

Reduction example

$$\begin{aligned} & (\nu xy)(x[5] . x(a) . \mathbf{0} \parallel y(b) . y[b] . \mathbf{0}) \\ & \quad \downarrow \\ & (\nu xy)(x(a) . \mathbf{0} \parallel y[5] . \mathbf{0}) \\ & \quad \downarrow \\ & (\nu xy)(\mathbf{0} \parallel \mathbf{0}) \end{aligned}$$

Reduction example

$$(\nu xy)(x[5] . x(a) . \mathbf{0} \parallel y(b) . y[b] . \mathbf{0})$$

\downarrow

$$(\nu xy)(x(a) . \mathbf{0} \parallel y[5] . \mathbf{0})$$

\downarrow

$$(\nu xy)(\mathbf{0} \parallel \mathbf{0})$$

\equiv

$$\mathbf{0} \parallel \mathbf{0}$$

Reduction example

$$\begin{aligned} & (\nu xy)(x[5] . x(a) . \mathbf{0} \parallel y(b) . y[b] . \mathbf{0}) \\ & \quad \downarrow \\ & (\nu xy)(x(a) . \mathbf{0} \parallel y[5] . \mathbf{0}) \\ & \quad \downarrow \\ & (\nu xy)(\mathbf{0} \parallel \mathbf{0}) \\ & \quad \equiv \\ & \mathbf{0} \parallel \mathbf{0} \\ & \quad \equiv \\ & \mathbf{0} \end{aligned}$$

Reduction rules

Comm. $(\nu xy)(x[V].P \parallel y(z).Q) \longrightarrow (\nu xy)(P \parallel Q\{V/z\})$

Reduction rules

$$\text{Comm.} \quad (\nu xy)(x[V].P \parallel y(z).Q) \longrightarrow (\nu xy)(P \parallel Q\{V/z\})$$

$$\text{Sel.} \quad (\nu xy)(x \triangleleft \ell_j.P \parallel y \triangleright \{\ell_i . Q_i\}_{i \in I}) \longrightarrow (\nu xy)(P \parallel Q_j) \quad (j \in I)$$

Reduction rules

$$\text{Comm.} \quad (\nu xy)(x[V].P \parallel y(z).Q) \longrightarrow (\nu xy)(P \parallel Q\{V/z\})$$

$$\text{Sel.} \quad (\nu xy)(x \triangleleft \ell_j.P \parallel y \triangleright \{\ell_i . Q_i\}_{i \in I}) \longrightarrow (\nu xy)(P \parallel Q_j) \quad (j \in I)$$

$$\frac{P \longrightarrow Q}{(\nu xy)P \longrightarrow (\nu xy)Q}$$

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

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

Lemma (Substitution)

If $\Gamma_1, x : A \vdash P$, and $\Gamma_2 \vdash V : A$, and Γ_1, Γ_2 is defined, then $\Gamma_1, \Gamma_2 \vdash P\{V/x\}$.

Proof.

By induction on the derivation of $\Gamma, x : A \vdash P$.



Lemma (Substitution)

If $\Gamma_1, x : A \vdash P$, and $\Gamma_2 \vdash V : A$, and Γ_1, Γ_2 is defined, then $\Gamma_1, \Gamma_2 \vdash P\{V/x\}$.

Proof.

By induction on the derivation of $\Gamma, x : A \vdash P$.



Lemma (Preservation (Structural congruence))

If $\Gamma \vdash P$ and $P \equiv Q$, then $\Gamma \vdash Q$

Proof.

By induction on the derivation of $P \equiv Q$.



Theorem (Preservation (Reduction))

If $\Gamma \vdash P$ and $P \longrightarrow Q$, then $\Gamma \vdash Q$.

Theorem (Preservation (Reduction))

If $\Gamma \vdash P$ and $P \longrightarrow Q$, then $\Gamma \vdash Q$.

Proof.

By induction on the derivation of $P \longrightarrow Q$.



Progress / Deadlock freedom?

...kind of. We will make this much stronger for a λ -calculus.

We get a weak result: within each session, as long as each process only communicates along the given endpoint, either the session can progress or has completed.

A Session-Typed Programming Language

From a process calculus to a programming language

A process calculus captures the essence of concurrent computation

- Concentrates only on communication
- Particularly useful for showing process equivalences

However, a programming language is much more complex

- Functions, let-bindings, (potentially nested) evaluation contexts...

How do we include session types in a programming language design?

Session types in a linear λ -calculus

Linear type theory for asynchronous session types

SIMON J. GAY

*Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK
(e-mail: simon@dcsc.gla.ac.uk)*

VASCO T. VASCONCELOS

*Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, 1749-016 Lisboa, Portugal
(e-mail: vv@di.fc.ul.pt)*

Abstract

Session types support a type-theoretic formulation of structured patterns of communication, so that the communication behaviour of agents in a distributed system can be verified by static typechecking. Applications include network protocols, business processes and operating system services. In this paper we define a multithreaded functional language with session types, which unifies, simplifies and extends previous work. There are four main contributions. First is an operational semantics with buffered channels, instead of the synchronous communication of previous work. Second, we prove that the session type of a channel gives an upper bound on the necessary size of the buffer. Third, session types are manipulated by means of the standard structures of a linear type theory, rather than by means of new forms of typing judgement. Fourth, a notion of subtyping, including the standard subtyping relation for session types (imported into the functional setting), and a novel form of subtyping between standard and linear function types, which allows the typechecker to handle linear types conveniently. Our new approach significantly simplifies session types in the functional setting, clarifies their essential features and provides a secure foundation for language developments such as polymorphism and object-orientation.

→ Idea: build session type constructs on top of the **linear λ -calculus**

→ The paper also shows mechanisms for session establishment (access points) and asynchronous communication

Example again: 2-Factor Authentication Server

```
twoFactorServer : TwoFactorServer  $\multimap$  ()  
twoFactorServer(s)  $\triangleq$   
  let ((username, password), s) = receive s in  
    if checkDetails(username, password) then  
      let s = select Authenticated s in serverMain(s)  
    else  
      let s = select AccessDenied s in close s
```

Let's build a session-typed λ -calculus (terms)

| | |
|------------|---|
| Types | $A, B ::= A \multimap B \mid A \times B \mid A + B \mid \mathbf{1} \mid C$ |
| Base types | $C ::= \text{Int} \mid \text{Bool} \mid \dots$ |
| Terms | $L, M, N ::= x \mid c$ $\lambda x. M \mid M N$ $() \mid \mathbf{let} () = M \mathbf{in} N$ $(M, N) \mid \mathbf{let} (x, y) = M \mathbf{in} N$ $\mathbf{inl} M \mid \mathbf{inr} M \mid \mathbf{case} M \{ \mathbf{inl} x \mapsto M; \mathbf{inr} y \mapsto N \}$ |

Let's build a session-typed λ -calculus (terms)

| | |
|---------------|--|
| Types | $A, B ::= A \multimap B \mid A \times B \mid A + B \mid \mathbf{1} \mid C$ |
| Session types | $S, T ::= !A.S \mid ?A.S \mid \text{end}$ |
| Base types | $C ::= \text{Int} \mid \text{Bool} \mid \dots$ |
| Terms | $L, M, N ::= x \mid c \mid ()$ $\mid \lambda x.M \mid M N$ $\mid () \mid \text{let } () = M \text{ in } N$ $\mid (M, N) \mid \text{let } (x, y) = M \text{ in } N$ $\mid \text{inl } M \mid \text{inr } M \mid \text{case } M \{ \text{inl } x \mapsto M; \text{inr } y \mapsto N \}$ $\mid \text{spawn } M \mid \text{new}$ $\mid \text{send } M N \mid \text{receive } M \mid \text{close } M$ |

Session Typing Constructs

$$\frac{\Gamma \vdash M : 1}{\Gamma \vdash \mathbf{spawn} M : 1}$$

Spawn M as a separate process.

Session Typing Constructs

$$\frac{\Gamma \vdash M : 1}{\Gamma \vdash \mathbf{spawn} M : 1}$$

Spawn M as a separate process.

$$\frac{}{\cdot \vdash \mathbf{new} : S \times \overline{S}}$$

Create a new channel, returning both endpoints.

Session Typing Constructs

$$\frac{\Gamma \vdash M : 1}{\Gamma \vdash \mathbf{spawn} M : 1}$$

Spawn M as a separate process.

$$\frac{}{\cdot \vdash \mathbf{new} : S \times \overline{S}}$$

Create a new channel, returning both endpoints.

$$\frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : !A.S}{\Gamma_1, \Gamma_2 \vdash \mathbf{send} M N : S}$$

Send a value M along session-typed channel endpoint N , returning the channel continuation.

Session Typing Constructs

$$\frac{\Gamma \vdash M : 1}{\Gamma \vdash \mathbf{spawn} M : 1}$$

Spawn M as a separate process.

$$\frac{}{\cdot \vdash \mathbf{new} : S \times \bar{S}}$$

Create a new channel, returning both endpoints.

$$\frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : !A.S}{\Gamma_1, \Gamma_2 \vdash \mathbf{send} M N : S}$$

Send a value M along session-typed channel endpoint N , returning the channel continuation.

$$\frac{\Gamma \vdash M : ?A.S}{\Gamma \vdash \mathbf{receive} M : A \times S}$$

Receive from endpoint M , returning received value and session continuation

Deadlock Freedom?

At present, we can still very easily deadlock:

```
let (x, y) = new in
let x = send 5 x in
let (z, x) = receive y in
close x; close y; z
```

Problem: Can construct **self-deadlocks** and **cyclic dependencies**

A solution (inspired by linear logic): combine spawning a process with creating a channel

- Restricts communication topologies to be **tree-shaped**
- Rules out both self-deadlocks and cyclic dependencies

Aside: Logical Correspondence

Session Types as Intuitionistic Linear Propositions

Luís Caires¹ and Frank Pfenning²

¹ CITI and Departamento de Informática, FCT, Universidade Nova de Lisboa

² Department of Computer Science, Carnegie Mellon University

Abstract. Several type disciplines for π -calculi have been proposed in which linearity plays a key role, even if their precise relationship with pure linear logic is still not well understood. In this paper we introduce a new system for this.

Propositions as Sessions

Philip Wadler
University of Edinburgh
wadler@ed.ac.uk

Abstract

Continuing a line of work by Abramsky (1994), by Bellare and Scott (1994), and by Caires and Pfenning (2010), among others, this paper presents CP, a calculus in which propositions of classical

logic (Hofmann 1997) introduced session types, further developed by Honda et al. (1998) and others, which take inspiration from linear logic, but do not enjoy a relationship as tight as Curry-Howard. Recently, Caires and Pfenning (2010) found a neat on Abramsky's observation that classical intuitionistic propositional

A Semantics for Propositions as Sessions

Sam Lindley and J. Garrett Morris

The University of Edinburgh
{Sam.Lindley, Garrett.Morris}@ed.ac.uk

Abstract. Session types provide a static guarantee that concurrent programs respect communication protocols. Recently, Caires, Pfenning, and Toninho, and Wadler, have developed a correspondence between propositions of linear logic and session typed π -calculus processes. We relate the cut-elimination semantics of this approach to an operational semantics for session-typed concurrency in a functional language. We begin by pre-

- **Caires & Pfenning** showed correspondence between session types and intuitionistic linear logic
- **Wadler** introduced two calculi, CP (process calculus based on CLL) and GV (session-typed linear λ -calculus)
- **Lindley & Morris** provided a semantics for GV, and semantics-preserving translations
- Not enough time to go into this in-depth today (ask if interested), main consequence: by keeping communication topology **tree-structured**, we get a well-behaved, canonical core calculus

Modifications

Split end into $\text{end}_!$ and $\text{end}_?$, and introduce **wait** to eliminate values of type $\text{end}_?$:

$$\frac{\Gamma \vdash M : \text{end}_?}{\Gamma \vdash \mathbf{wait} M : 1}$$

Fuse **spawn** and **new** into **fork**:

$$\frac{\Gamma \vdash M : S \multimap \text{end}_!}{\Gamma \vdash \mathbf{fork} M : \overline{S}}$$

What about branching and selection?

Branching and selection can be **encoded**:

$$\mathbf{select} \ell M \triangleq \mathbf{let} x = \mathbf{send} () M \mathbf{in} \\ \mathbf{fork} (\lambda y. \mathbf{send} (\ell y) x)$$

$$S \oplus T \triangleq !1 . !(\bar{S} + \bar{T}) . \mathbf{end}_! \\ S \& T \triangleq ?1 . ?(S + T) . \mathbf{end}_?$$

$$\mathbf{offer} L \{ \mathbf{inl} x \mapsto M, \mathbf{inr} y \mapsto N \} \triangleq \\ \mathbf{let} ((), z) = \mathbf{receive} L \mathbf{in} \\ \mathbf{let} (w, z) = \mathbf{receive} z \mathbf{in} \\ \mathbf{let} () = \mathbf{wait} z \mathbf{in} \\ \mathbf{case} w \{ \mathbf{inl} x \mapsto M, \mathbf{inr} y \mapsto N \}$$

Idea due to CPS translations by Dardha et al. (2012).

Reduction: Example

Fork a child thread and send an integer; forked thread sends back integer + 10

- $$\left(\begin{array}{l} \text{let } s = \text{fork} \left(\begin{array}{l} \lambda c. \\ \text{let } (x, c) = \text{receive } c \text{ in} \\ \text{send } (x + 10) c \end{array} \right) \text{ in} \\ \text{let } s = \text{send } 5 \text{ in} \\ \text{let } (x, s) = \text{receive } s \text{ in} \\ \text{wait } s; x \end{array} \right)$$

Reduction: Example

$$(\nu ab) \left(\bullet \left(\begin{array}{l} \text{let } s = a \text{ in} \\ \text{let } s = \text{send } 5 \text{ } s \text{ in} \\ \text{let } (x, s) = \text{receive } s \text{ in} \\ \text{wait } s; x \end{array} \right) \parallel \circ \left(\begin{array}{l} \text{let } (x, c) = \text{receive } b \text{ in} \\ \text{send } (x + 10) \text{ } c \end{array} \right) \right)$$

Reduction: Example

$$(\nu ab) \left(\bullet \left(\begin{array}{l} \text{let } s = \text{send } 5 \text{ a in} \\ \text{let } (x, s) = \text{receive } s \text{ in} \\ \text{wait } s; x \end{array} \right) \parallel \circ \left(\begin{array}{l} \text{let } (x, c) = \text{receive } b \text{ in} \\ \text{send } (x + 10) c \end{array} \right) \right)$$

Reduction: Example

$$(\nu ab) \left(\bullet \left(\begin{array}{l} \text{let } s = a \text{ in} \\ \text{let } (x, s) = \text{receive } s \text{ in} \\ \text{wait } s; x \end{array} \right) \parallel \circ \left(\begin{array}{l} \text{let } (x, c) = (5, b) \text{ in} \\ \text{send } (x + 10) c \end{array} \right) \right)$$

Reduction: Example

$$(\nu ab) \left(\bullet \left(\begin{array}{l} \text{let } (x, s) = \text{receive } a \text{ in} \\ \text{wait } s; x \end{array} \right) \parallel \circ \left(\text{send } (5 + 10) b \right) \right)$$

Reduction: Example

$$(\nu ab) \left(\bullet \left(\begin{array}{l} \text{let } (x, s) = \text{receive } a \text{ in} \\ \text{wait } s; x \end{array} \right) \parallel \circ \left(\text{send } 15 \text{ } b \right) \right)$$

Reduction: Example

$$(\nu ab) \left(\bullet \left(\begin{array}{l} \mathbf{let} (x, s) = (15, a) \mathbf{in} \\ \mathbf{wait} s; x \end{array} \right) \parallel \circ(b) \right)$$

Reduction: Example

$(\nu ab) (\bullet (\mathbf{wait} \ a; 15) \parallel \circ (b))$

Reduction: Example

• (15)

Runtime Syntax

Thread flags

$\phi ::= \bullet \mid \circ$

Runtime Syntax

Thread flags

$\phi ::= \bullet \mid \circ$

Runtime Syntax

Thread flags

Configurations

$$\begin{aligned}\phi &::= \bullet \mid \circ \\ \mathcal{C}, \mathcal{D} &::= (\nu xy)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \phi M\end{aligned}$$

Runtime Syntax

Thread flags

$$\phi ::= \bullet \mid \circ$$

Configurations

$$\mathcal{C}, \mathcal{D} ::= (\nu xy)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \phi M$$

Values

$$V, W ::= c \mid () \mid \lambda x.M \mid (V, W) \mid \mathbf{inl} V \mid \mathbf{inr} W$$

Runtime Syntax

Thread flags

$\phi ::= \bullet \mid \circ$

Configurations

$\mathcal{C}, \mathcal{D} ::= (\nu xy)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \phi M$

Values

$V, W ::= c \mid () \mid \lambda x.M \mid (V, W) \mid \mathbf{inl} V \mid \mathbf{inr} W$

Evaluation contexts

$E ::= [] \mid E M \mid V E$
| **let** $() = E$ **in** M
| $(E, M) \mid (V, E) \mid \mathbf{let} (x, y) = E$ **in** M
| **inl** $E \mid \mathbf{inr} E \mid \mathbf{case} E \{ \mathbf{inl} x \mapsto M; \mathbf{inr} y \mapsto N \}$
| **fork** $E \mid \mathbf{send} E M \mid \mathbf{send} V E$
| **receive** $E \mid \mathbf{close} E$

Runtime Syntax

| | |
|---------------------|---|
| Thread flags | $\phi ::= \bullet \mid \circ$ |
| Configurations | $\mathcal{C}, \mathcal{D} ::= (\nu xy)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \phi M$ |
| Values | $V, W ::= c \mid () \mid \lambda x.M \mid (V, W) \mid \mathbf{inl} V \mid \mathbf{inr} W$ |
| Evaluation contexts | $E ::= [] \mid E M \mid V E$ let $() = E$ in M $(E, M) \mid (V, E) \mid \mathbf{let} (x, y) = E$ in M inl $E \mid \mathbf{inr} E \mid \mathbf{case} E \{ \mathbf{inl} x \mapsto M; \mathbf{inr} y \mapsto N \}$ fork $E \mid \mathbf{send} E M \mid \mathbf{send} V E$ receive $E \mid \mathbf{close} E$ |
| Thread contexts | $\mathcal{F} ::= \phi E$ |

Reduction rules

(Fork)

$$\mathcal{F}[\mathbf{fork} \lambda z.M] \longrightarrow (\nu xy)(\mathcal{F}[x] \parallel \circ M\{y/z\})$$

Reduction rules

(Fork) $\mathcal{F}[\mathbf{fork} \lambda z.M] \longrightarrow (\nu xy)(\mathcal{F}[x] \parallel \circ M\{y/z\})$

(Comm) $(\nu xy)(\mathcal{F}[\mathbf{send} V x] \parallel \mathcal{F}'[\mathbf{receive} y]) \longrightarrow (\nu xy)(\mathcal{F}[x] \parallel \mathcal{F}'[(V, y)])$

Reduction rules

(Fork) $\mathcal{F}[\mathbf{fork} \lambda z.M] \longrightarrow (\nu xy)(\mathcal{F}[x] \parallel \circ M\{y/z\})$

(Comm) $(\nu xy)(\mathcal{F}[\mathbf{send} V x] \parallel \mathcal{F}'[\mathbf{receive} y]) \longrightarrow (\nu xy)(\mathcal{F}[x] \parallel \mathcal{F}'[(V, y)])$

(Wait) $(\nu xy)(\mathcal{F}[\mathbf{wait} x] \parallel \circ y) \longrightarrow \mathcal{F}[()]$

Reduction rules

(Fork) $\mathcal{F}[\mathbf{fork} \lambda z.M] \longrightarrow (\nu xy)(\mathcal{F}[x] \parallel \circ M\{y/z\})$

(Comm) $(\nu xy)(\mathcal{F}[\mathbf{send} V x] \parallel \mathcal{F}'[\mathbf{receive} y]) \longrightarrow (\nu xy)(\mathcal{F}[x] \parallel \mathcal{F}'[(V, y)])$

(Wait) $(\nu xy)(\mathcal{F}[\mathbf{wait} x] \parallel \circ y) \longrightarrow \mathcal{F}[()]$

$$\frac{C \longrightarrow C'}{C \parallel D \longrightarrow C' \parallel D}$$

$$\frac{C \longrightarrow D}{(\nu xy)C \longrightarrow (\nu xy)D}$$

$$\frac{M \longrightarrow_{\beta} N}{\mathcal{F}[M] \longrightarrow \mathcal{F}[N]}$$

Metatheory?

So far, we have a syntax, static typing rules, and reduction semantics

To **prove** properties, we need typing rules for runtime environments

- Unlike term typing rules, these are for reasoning only and would not be implemented in a typechecker

Hyper-environments

To prove deadlock-freedom / progress, we need some way of recording that two endpoints of a channel must be used in separate threads

A **hyper-environment** is a set of type environments; each environment must be used to type a different thread

$$\mathcal{G} ::= \Gamma_1 \parallel \cdots \parallel \Gamma_n$$

Idea inspired by **hypersequent logics** (Avron, 1991)

Investigated for CP by Kokke, Montesi & Peressotti (2018, 2019); for GV by F. et al. (2021, 2023)

Runtime types

Runtime types $R ::= \bullet A \mid \circ$

Can only have a **single** main thread to return a value

$$\bullet A \sqcap \circ = \bullet A$$

$$\circ \sqcap \bullet A = \bullet A$$

$$\circ \sqcap \circ = \circ$$

$\bullet A \sqcap \bullet B$ undefined

Configuration typing

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \bullet M : \bullet A}$$

Main thread M of type A , typable under singleton environment

Configuration typing

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \bullet M : \bullet A}$$

Main thread M of type A , typable under singleton environment

$$\frac{\Gamma \vdash M : \text{end}_l}{\Gamma \vdash \circ M : \circ}$$

Child thread M of type end_l , typable under singleton environment

Configuration typing

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \bullet M : \bullet A}$$

Main thread M of type A , typable under singleton environment

$$\frac{\Gamma \vdash M : \text{end}_l}{\Gamma \vdash \circ M : \circ}$$

Child thread M of type end_l , typable under singleton environment

$$\frac{\mathcal{G} \parallel \Gamma, x : S \parallel \Gamma_2, y : \bar{S} \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \vdash (\nu xy)\mathcal{C} : R}$$

Bind co-names x and y with dual types in \mathcal{C} , introducing a separator between them

Configuration typing

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \bullet M : \bullet A}$$

Main thread M of type A , typable under singleton environment

$$\frac{\Gamma \vdash M : \text{end}_l}{\Gamma \vdash \circ M : \circ}$$

Child thread M of type end_l , typable under singleton environment

$$\frac{\mathcal{G} \parallel \Gamma, x : S \parallel \Gamma_2, y : \bar{S} \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \vdash (\nu xy)\mathcal{C} : R}$$

Bind co-names x and y with dual types in \mathcal{C} , introducing a separator between them

$$\frac{\mathcal{G} \vdash \mathcal{C} : R_1 \quad \mathcal{H} \vdash \mathcal{D} : R_2}{\mathcal{G} \parallel \mathcal{H} \vdash \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2}$$

Configurations \mathcal{C} and \mathcal{D} typable under separated hyper-environments, with combinable runtime types

Preservation

Lemma (Preservation (\equiv))

If $\mathcal{G} \vdash \mathcal{C} : R$ and $\mathcal{C} \equiv \mathcal{D}$, then $\mathcal{G} \vdash \mathcal{D} : R$.

Proof.

By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$.



Preservation

Lemma (Preservation (\equiv))

If $\mathcal{G} \vdash \mathcal{C} : R$ and $\mathcal{C} \equiv \mathcal{D}$, then $\mathcal{G} \vdash \mathcal{D} : R$.

Proof.

By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$. □

Theorem (Preservation (\longrightarrow))

If $\mathcal{G} \vdash \mathcal{C} : R$ and $\mathcal{C} \longrightarrow \mathcal{D}$, then $\mathcal{G} \vdash \mathcal{D} : R$.

Proof.

By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{D}$, using the above lemma, a substitution lemma, and lemmas for manipulating evaluation contexts. □

Proving progress (1): Canonical Forms

Tree canonical form

$$(\nu x_1 y_1)(\circ M_1 \parallel \dots \parallel (\nu x_n y_n)(\circ M_n \parallel \bullet N) \dots)$$

(where $x_i \in \text{fv}(M_i)$ for each $i \in 1..n$)

- Makes essential use of hyper-environment separation, as well as **abstract process structures** to show processes must be a tree

Proving progress (2)

Open progress

- **Blocked**: wishing to perform an action on a name
- Each thread must be blocked on a free variable or a previous ν -bound name

Proving progress (2)

Open progress

- **Blocked**: wishing to perform an action on a name
- Each thread must be blocked on a free variable or a previous ν -bound name

Theorem (Global Progress)

If $\cdot \vdash \mathcal{C} : \bullet A$ where \mathcal{C} is in canonical form and A does not contain session types, then either $\mathcal{C} = \bullet V$ for some V , or $\mathcal{C} \longrightarrow$.

Proving progress (2)

Open progress

- **Blocked**: wishing to perform an action on a name
- Each thread must be blocked on a free variable or a previous ν -bound name

Theorem (Global Progress)

If $\cdot \vdash \mathcal{C} : \bullet A$ where \mathcal{C} is in canonical form and A does not contain session types, then either $\mathcal{C} = \bullet V$ for some V , or $\mathcal{C} \longrightarrow$.

Proof sketch.

- Consider the case where \mathcal{C} cannot reduce: in this case all threads must be values or blocked.
- Since \mathcal{C} is closed, either $M_1 = \circ x_1$ or must be blocked on x_1 ; thread $\circ M_2$ cannot be blocked on y_1 (as by duality reduction would occur) so must be $\circ x_2$ or blocked on x_2 , and so on.
- Since the type of the main thread does not contain session types, $\text{fv}(N) = \emptyset$ and so it follows that no threads can be blocked and main thread must be a value.

Diamond Property

- Any two possible reductions can be reconciled in a single step

Strong Normalisation

- No infinite reduction sequences
- Simple proof due to linearity; extensions need more work

Session types in practice: An implementation in Links

Links

<http://www.links-lang.org>

Linking theory to practice
for the web



Database
Integration



Relational
Lenses

Temporal
Data
Management

Language-
Integrated
Query

Provenance

Typed
HTML

Formlets

Model-
View-
Update

Web Development



Concurrency &
Distribution



RPC
Calculus

Distributed
Session
Types

Session
Exceptions

Actor-style
concurrency



Types

First-class
Polymorphism

Row Types

Linear Types

Effect Handlers



CEK
Machine
(Server)

CPS
Translation
(Client)

Row-based
Effects

WASM
Backend



Session types in Links

Links has an implementation of session types including recursion, polymorphism, asynchrony...

Session types integrate smoothly with standard Hindley-Milner type inference

Based on an extension of System F (FST), where kinds $K(Y, Z)$ include:

- A **primary kind** K : either **type**, **row**, or **presence**
- A **linearity** Y : either **linear** or **unrestricted**
- A **restriction** Z : either **any** or **session**

Demo

Exceptions

Links is a **web language**. Client may close their browser in the middle of a session!

We need to be able to safely dispose of (linear) session endpoints:

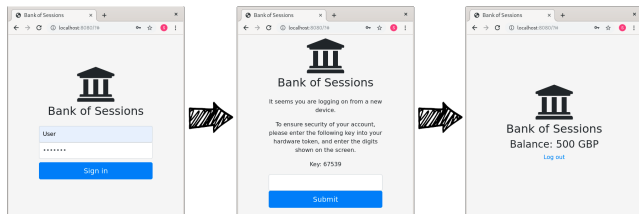
$$\frac{\Gamma \vdash M : S}{\Gamma \vdash \mathbf{cancel} M : 1} \qquad \frac{}{\cdot \vdash \mathbf{raise} : A}$$
$$\frac{\Gamma_1 \vdash L : A \quad \Gamma_2, x : A \vdash M : B \quad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash \mathbf{try} L \mathbf{as} x \mathbf{in} M \mathbf{otherwise} N : B}$$

cancel allows us to throw away an endpoint; interaction with cancelled endpoint raises an exception

Exceptions cancel all endpoints in aborted continuation

Exceptional Asynchronous Session Types, POPL'19; ideas based on those of Mostrous & Vasconcelos (2014)

Integrating Session Types with GUI Programming



Problem: Linearity difficult to reconcile with asynchrony of GUI programming

- Clicking a button twice: sending a message twice!
- Most session-typed programs are command-line applications

Idea: Extend widely-used GUI framework called Model-View-Update to support linear resources

- **Cancellation:** Useful for safely discarding (possibly-linear) messages from previous states

Model-View-Update-Communicate: Session Types Meet the Elm Architecture (ECOOP 2020)

Wrapping up

Conclusion

Today:

- Session types, by example

- A session-typed process calculus

- Scaling to a deadlock-free programming language

- Practical implementation in Links

Next time: Multiparty session types

Questions?