Event-Driven Multiparty Session Actors

Simon Fowler & Raymond Hu

HOPE, 4th September 2023

Engineering and Physical Sciences Research Council

University of Glasgow

Queen Mary University of London

Actor Languages

- Actor languages: **communication-centric** languages
  - Communication via **explicit message passing**
- Popular for writing reliable distributed code
  - Power WhatsApp, with >1B users
  - Support **data locality** and idioms such as **supervision hierarchies**

Communication Errors

Communication Mismatch

- Receiving an unexpected message / message with unexpected payload type

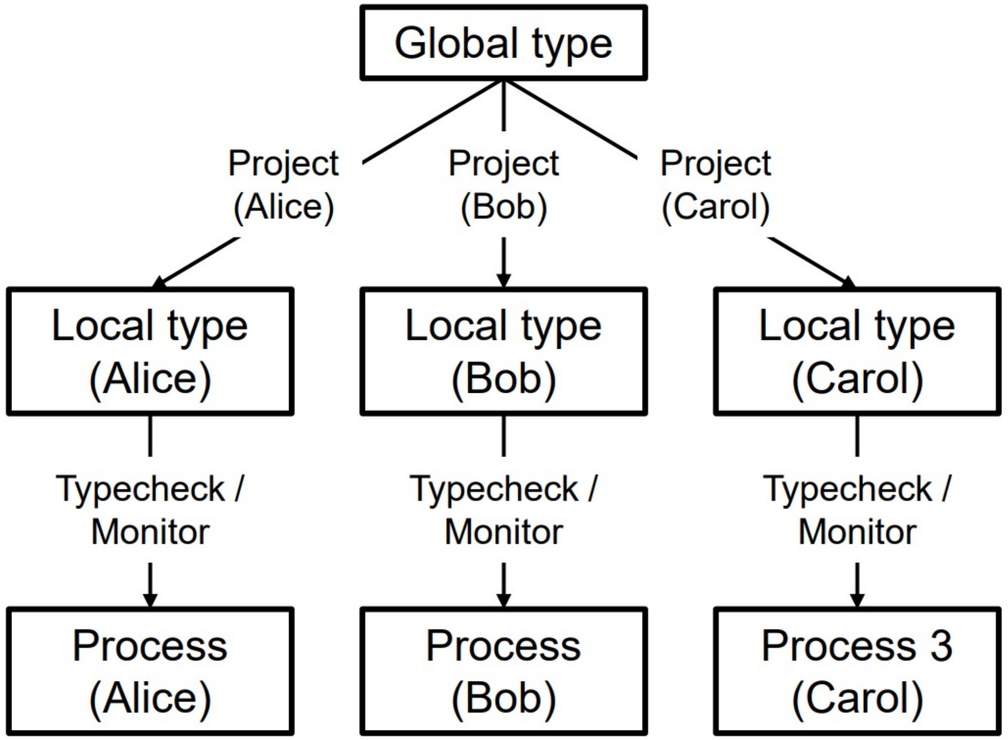
Deadlock

- Cyclic dependencies meaning communication cannot proceed

Livelock

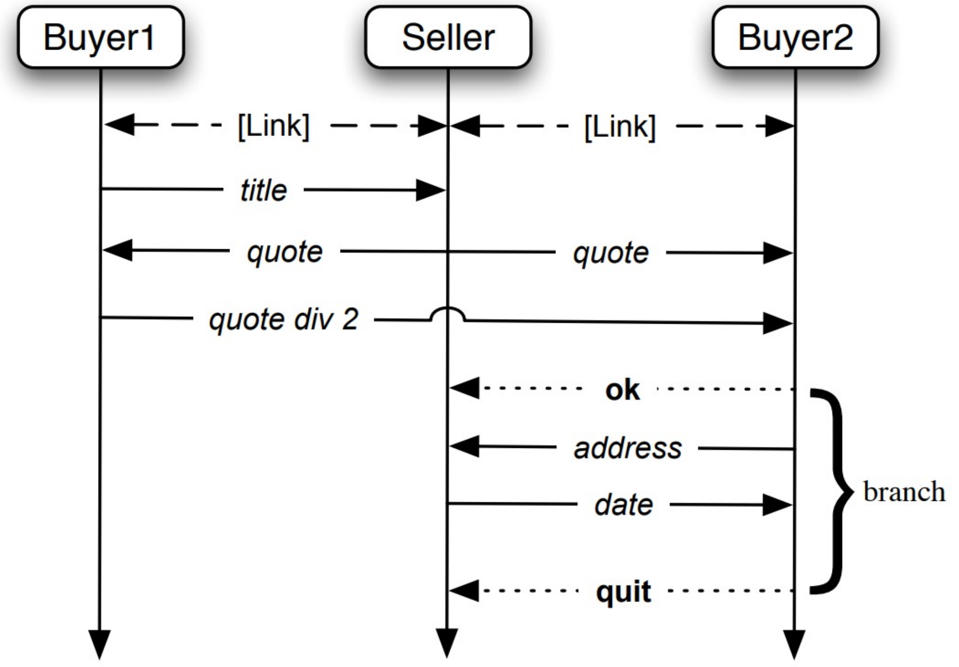
- Communication in one part of the system starves communication in another part of the system

Multiparty Session Types



- Session types: **types for protocols**
    - Goal is to rule out communication errors statically
- Interactions between participants specified as a **global type**
- Global types projected to **local types** for each participant
- Local types used to **typecheck** or **monitor** individual processes

Example: Two-Buyer Protocol



- Two buyers co-ordinate to buy an expensive item
  - Traditionally a textbook, apt given we're in the US
- Buyer1 asks for quote from seller
- Buyer1 contacts Buyer2 with their share
- Buyer2 can then either accept or reject

$$
\begin{array}{l}
\text{Buyer1} \longrightarrow \text{Seller} : \texttt{title}(\texttt{String}) \,.\\
\text{Seller} \longrightarrow \text{Buyer1} : \texttt{quote}(\texttt{Int}) \,.\\
\text{Buyer1} \longrightarrow \text{Buyer2} : \texttt{share}(\texttt{Int}) \,.\\
\text{Buyer2} \longrightarrow \text{Seller} : \{\\
\quad \texttt{address}(\texttt{String}) \,.\\
\qquad \text{Seller} \longrightarrow \text{Buyer2} : \texttt{date}(\texttt{Date}) \,.\, \texttt{end},\\
\quad \texttt{quit}(\texttt{Unit}) \,.\, \texttt{end}\\
\}
\end{array}
$$

$\upharpoonright$ Buyer1

Buyer1

$$
\begin{array}{l}
\text{Seller} \oplus \texttt{title}(\texttt{String}) \,.\, \text{Seller} \,\&\, \texttt{quote}(\texttt{String}) \,.\\
\text{Buyer2} \oplus \texttt{share}(\texttt{Int}) \,.\, \texttt{end}
\end{array}
$$

$\upharpoonright$ Buyer2

Buyer2

$$
\begin{array}{l}
\text{Buyer1} \,\&\, \texttt{share}(\texttt{Int}) \,.\, \text{Seller} \oplus \{\\
\quad \texttt{address}(\texttt{String}) \,.\, \text{Seller} \,\&\, \texttt{date}(\texttt{Date}) \,.\, \texttt{end}\\
\quad \texttt{quit}(\texttt{Unit}) \,.\, \texttt{end} \,\}
\end{array}
$$

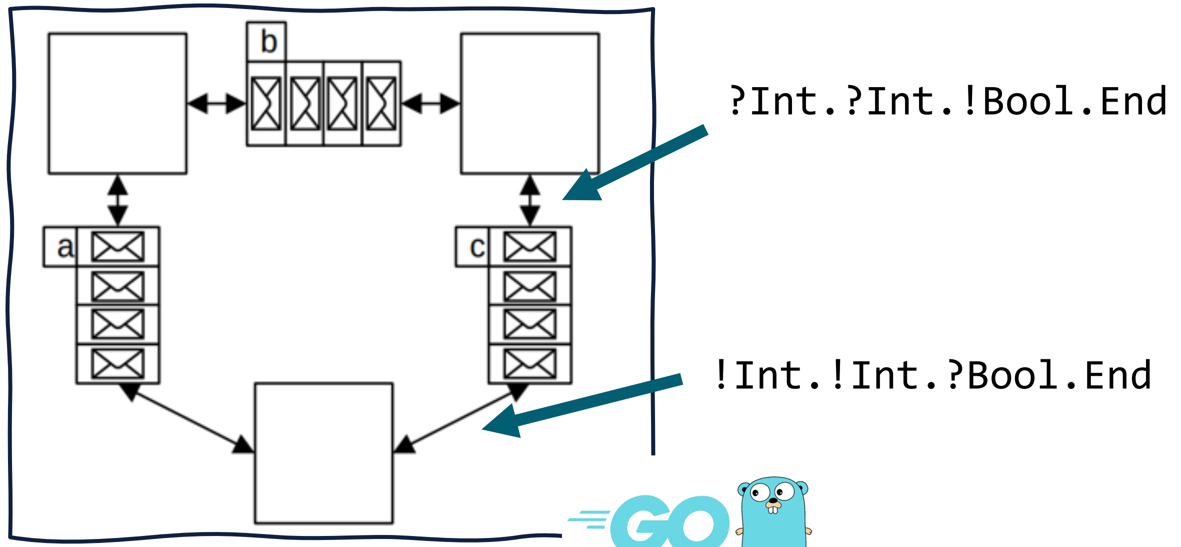
$\upharpoonright$ Seller

Seller

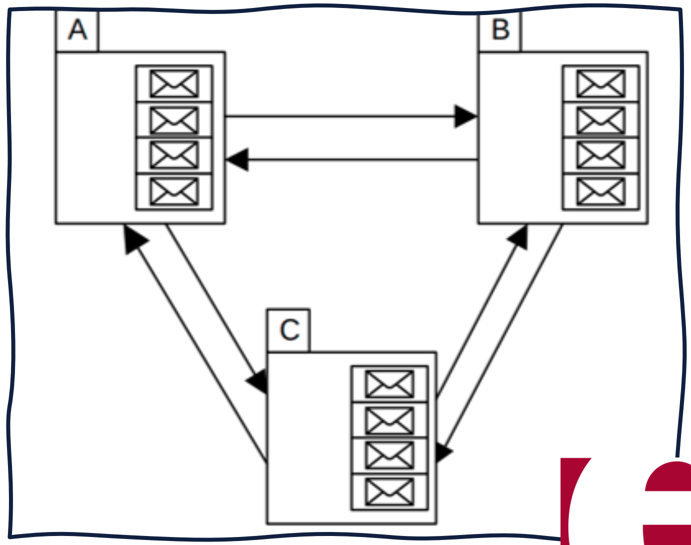
$$
\begin{array}{l}
\text{Buyer1} \,\&\, \texttt{title}(\texttt{String}) \,.\, \text{Buyer1} \oplus \texttt{quote}(\texttt{Int}) \,.\\
\text{Buyer2} \,\&\{\\
\quad \texttt{address}(\texttt{String}) \,.\, \text{Buyer2} \oplus \texttt{date}(\texttt{Date}) \,.\, \texttt{end}\\
\quad \texttt{quit}(\texttt{Unit}) \,.\, \texttt{end} \,\}
\end{array}
$$

How can we apply session types to actor languages?

Channels vs. Actors



Channels



Actors

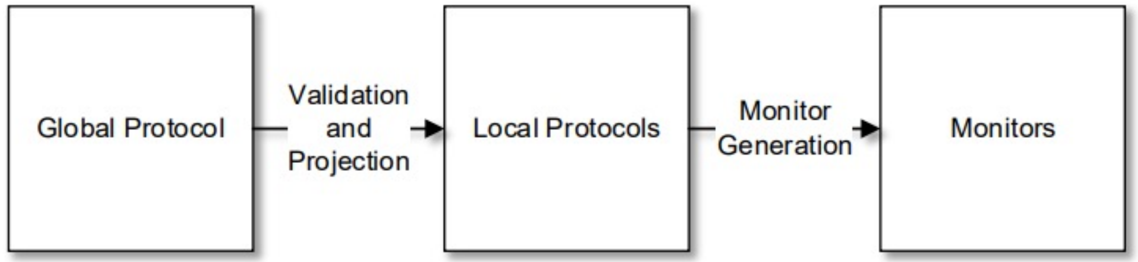
Named buffers, anonymous processes
Easy to type
Distributed programming difficult
- Choice requires distributed algorithms
- No locality: distributed delegation tricky

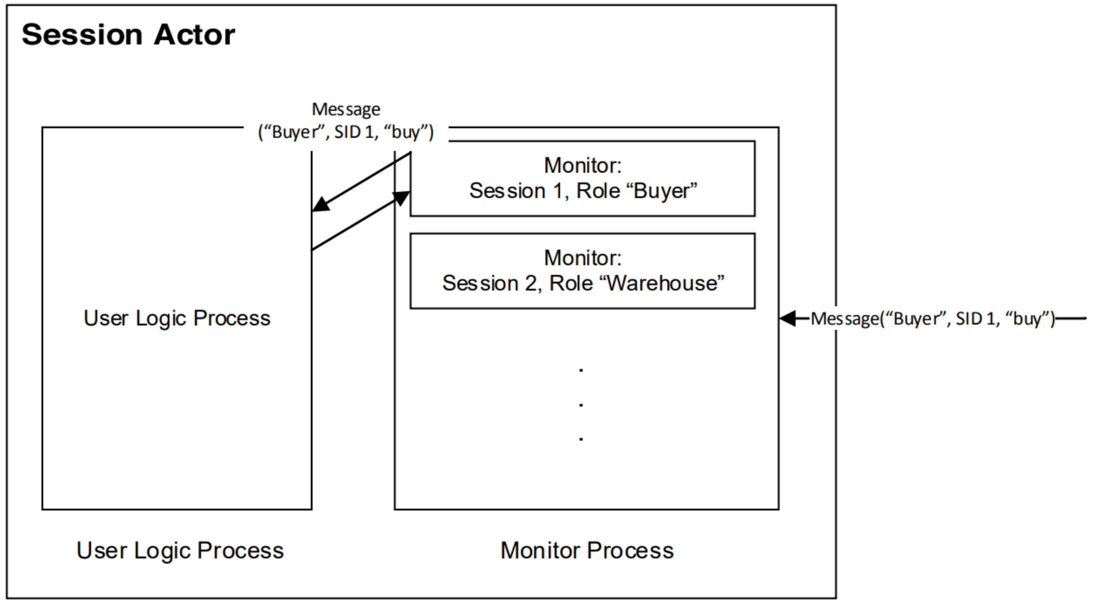
Named processes, point-to-point messaging
More difficult to type
Distributed programming much easier
- Messages always local to process
- Named processes allow supervision

Type the **mailbox**

Type the **process**

Multiparty Session Actors




Session Actor
Message (“Buyer”, SID 1, “buy”)
Monitor: Session 1, Role “Buyer”
Monitor: Session 2, Role “Warehouse”
User Logic Process
Message(“Buyer”, SID 1, “buy”)
User Logic Process
Monitor Process

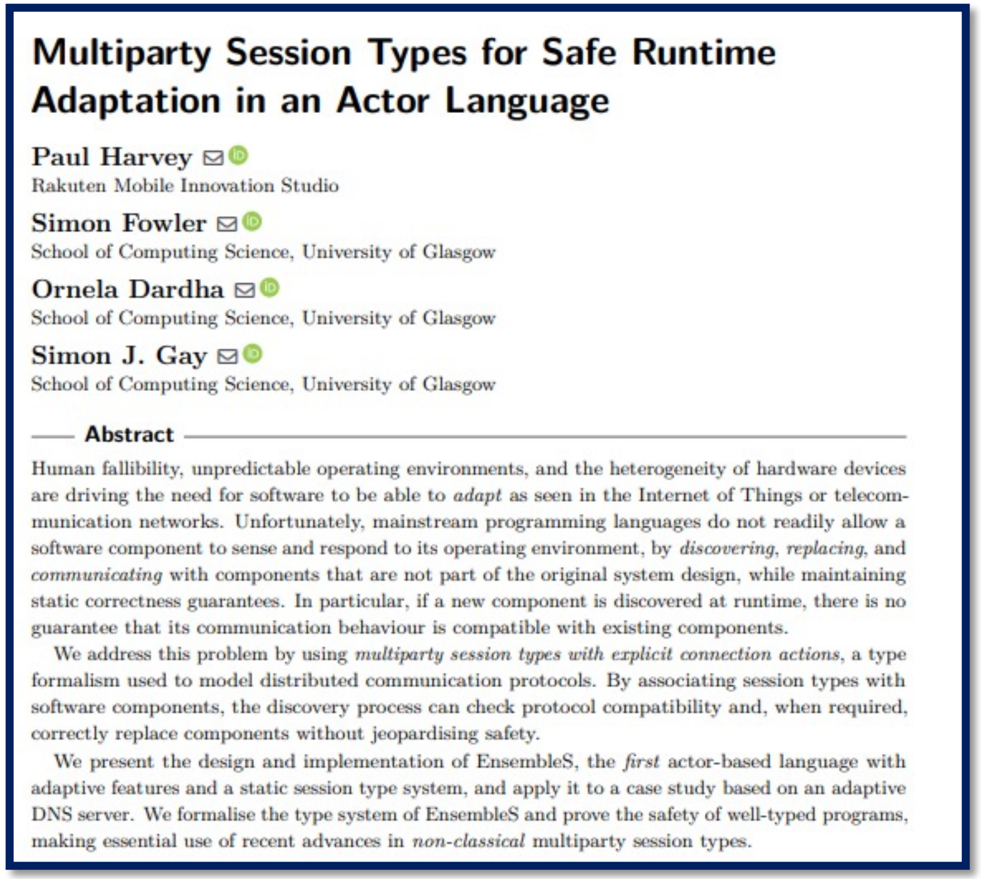
Idea: Actor can take part in multiple roles, in multiple sessions

Main benefit: sessions can share state

Messages pass through **runtime monitor** to check against session type

Originally due to Neykova & Yoshida, COORDINATION’14

Static Checking?



Multiparty Session Types for Safe Runtime Adaptation in an Actor Language

Paul Harvey
Rakuten Mobile Innovation Studio

Simon Fowler
School of Computing Science, University of Glasgow

Ornela Dardha
School of Computing Science, University of Glasgow

Simon J. Gay
School of Computing Science, University of Glasgow

Abstract

Human fallibility, unpredictable operating environments, and the heterogeneity of hardware devices are driving the need for software to be able to *adapt* as seen in the Internet of Things or telecommunication networks. Unfortunately, mainstream programming languages do not readily allow a software component to sense and respond to its operating environment, by *discovering*, *replacing*, and *communicating* with components that are not part of the original system design, while maintaining static correctness guarantees. In particular, if a new component is discovered at runtime, there is no guarantee that its communication behaviour is compatible with existing components.

We address this problem by using *multiparty session types with explicit connection actions*, a type formalism used to model distributed communication protocols. By associating session types with software components, the discovery process can check protocol compatibility and, when required, correctly replace components without jeopardising safety.

We present the design and implementation of EnsembleS, the *first* actor-based language with adaptive features and a static session type system, and apply it to a case study based on an adaptive DNS server. We formalise the type system of EnsembleS and prove the safety of well-typed programs, making essential use of recent advances in *non-classical* multiparty session types.

Use **flow-sensitive** effect-typing judgement to enforce session typing

$$\Gamma \mid S \triangleright M : A \triangleleft T$$

Pre- and post-conditions enforce session typing

Limitation: Only **one** session at a time

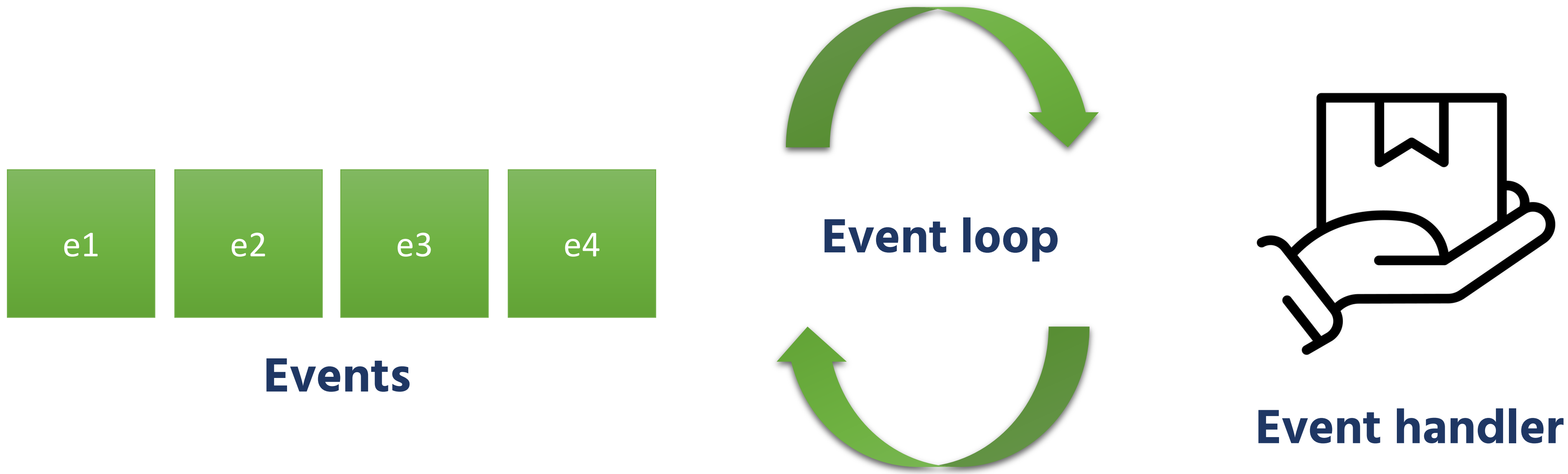
Dynamic checking, multiple sessions

or

Static checking, single session

Key idea: Combine a **flow-sensitive effect system** with **event-driven programming**

Event-driven programming



Isolated Processes

Co-ordination only through message passing

First-class Event Handlers

Event handlers are values (can even be passed in messages)

Flow-Sensitive Effect Typing

Enforce session typing by governing the actions that can be performed

Access Points

Actors register to take part in sessions

Example: Two-Buyer Protocol

Create access point to allow two-buyer sessions to be established

$$\begin{array}{l}
\text{main} \triangleq \\
\quad \textbf{let } ap \Leftarrow \textbf{newAP}_{(\text{Seller}:\text{S},\text{Buyer1}:\text{B1},\text{Buyer2}:\text{B2})} \textbf{ in} \\
\quad \textbf{spawn } (\text{seller}(ap)\ ()); \\
\quad \text{spawnBuyers}(ap, \text{“}\texttt{Types and Programming Languages}\text{”}); \\
\quad \text{spawnBuyers}(ap, \text{“}\texttt{Compiling with Continuations}\text{”}) \\
\\
\text{spawnBuyers}(ap, title) \triangleq \\
\quad \textbf{spawn } \text{buyer1}(ap, title);\ \textbf{spawn } \text{buyer2}(ap)
\end{array}$$

Spawn a seller, and two sets of buyers, all given AP reference

Example: Seller

$$\mathsf{seller}(ap) \triangleq$$

$$\mathbf{register}\ ap\ \mathsf{Seller}\ \left(\left(\begin{array}{l}\mathbf{rec}\ install(\_).\\ \quad \mathbf{register}\ ap\ \mathsf{Seller}\ (install\ ());\\ \quad \mathbf{suspend}\ \mathsf{titleHandler}\end{array}\right)\ ()\right)$$

Register to take part as Seller. When session established, callback will recursively register, then suspend awaiting title

$$\mathsf{titleHandler} \triangleq$$

$$\begin{array}{l}\mathbf{handler}\ \mathsf{Buyer1}\ \{\\ \quad \mathtt{title}(x) \mapsto\\ \qquad \mathsf{Buyer1}\,!\,\mathsf{quote}(\mathsf{lookupPrice}(x));\ \mathbf{suspend}\ \mathsf{decisionHandler}\ \}\end{array}$$

Invoked when message has been received from Buyer1. Sends quote to Buyer1, suspends awaiting decision from Buyer2

$$\mathsf{decisionHandler} \triangleq$$

$$\begin{array}{l}\mathbf{handler}\ \mathsf{Buyer2}\ \{\\ \quad \mathtt{address}(addr) \mapsto \mathsf{Buyer2}\,!\,date(\mathsf{shippingDate}(addr)),\\ \quad \mathtt{quit}(\_) \mapsto \mathbf{return}\ ()\ \}\end{array}$$

Invoked when decision has been received from Buyer2. Sends date to Buyer2, or terminates

$$
\begin{array}{llcl}
\text{Roles} & \mathsf{p}, \mathsf{q} & & \\
\text{Variables} & x, y, z, f & & \\
\text{Values} & V, W & ::= & x \mid \lambda x.M \mid \textbf{rec}\ f(x).M \mid c \mid \textbf{handler}\ \mathsf{p}\ \{\overrightarrow{H}\} \\
\text{Message Handlers} & H & ::= & \ell(x) \mapsto M \\
\text{Computations} & L, M, N & ::= & \textbf{let}\ x \Leftarrow M\ \textbf{in}\ N \mid \textbf{return}\ V \mid V\ W \\
 & & \mid & \textbf{if}\ V\ \textbf{then}\ M\ \textbf{else}\ N \\
 & & \mid & \textbf{spawn}\ M \mid \mathsf{p}\,!\,\ell(V) \mid \textbf{suspend}\ V \\
 & & \mid & \textbf{newAP}_{(\mathsf{p}_i : T_i)_i} \mid \textbf{register}\ V\ \mathsf{p}\ M
\end{array}
$$

Explicit stratification of values and computations (fine-grain call-by-value)

$$\boxed{\Gamma \vdash V : A}$$

Under environment Γ, value V has type A

$$\boxed{\Gamma \mid S \triangleright M : A \triangleleft T}$$

Under environment Γ, with session pre-condition S, computation M has type A and post-condition T

$$\frac{j \in I \qquad \Gamma \vdash V : A_j}{\Gamma \mid \mathsf{p} \oplus \{\ell_i(A_i).S_i\}_{i \in I} \triangleright \mathsf{p} \,!\, \ell_j(V) : \mathbf{1} \triangleleft S_j}$$

Send:

Precondition must permit us sending ℓ_j, payload type must match
Postcondition depends on which message we send

$$\frac{(\Gamma, x_i : A_i \mid S_i \triangleright M_i : \mathbf{1} \triangleleft \mathsf{end})_i}{\Gamma \vdash \mathbf{handler}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\} : \mathsf{Handler}(\mathsf{p}\ \&\{\ell_i(A_i).S_i\}_i)}$$

Handlers:

Not an effect handler!

First-class representation of an event handler

Parameterised by a **receive** session type

Each branch has a session continuation, must complete session or suspend

$$\frac{\Gamma \vdash V : \text{Handler}(S^{?})}{\Gamma \mid S^{?} \triangleright \textbf{suspend}\, V : A \triangleleft S'}$$

Suspend:
Installs given handler, returns to being idle

$$\frac{\Gamma \mid \text{end} \triangleright M : \mathbf{1} \triangleleft \text{end}}{\Gamma \mid S \triangleright \textbf{spawn}\, M : \mathbf{1} \triangleleft S}$$

Spawn:
Creates a new process, not in a session

$$\frac{\varphi \text{ is a safety property} \qquad \varphi((\mathsf{p}_i : T_i)_{i \in I})}{\Gamma \mid S \triangleright \mathbf{newAP}_{(\mathsf{p}_i : T_i)_{i \in I}} : \mathsf{AP}((\mathsf{p}_i : T_i)_{i \in I}) \triangleleft S}$$

newAP:
Creates an access point (collection of types must be *safe:* cannot exchange values of mismatching types)

$$\frac{\begin{array}{c} j \in I \\ \Gamma \vdash V : \mathsf{AP}((\mathsf{p}_i : T_i)_{i \in I}) \qquad \Gamma \mid T_j \triangleright M : \mathbf{1} \triangleleft \mathsf{end} \end{array}}{\Gamma \mid S \triangleright \mathbf{register}\ V\ \mathsf{p}_j\ M : \mathbf{1} \triangleleft S}$$

Register:
Registers with an access point to take part in a session

Semantics (Overview)

$$\langle T, \sigma, \rho \rangle$$

Process state, either:

- Idle
- Term M (not in a session)
- Term $M@s[p]$ (playing role p in session s)

Handler state:
Stores event handlers to be invoked when a message arrives

Initialisation state, used to establish sessions

Semantics formulated as concurrent lambda calculus; asynchronous communication model

Metatheory

- **Type preservation**
  - Reduction preserves typability
  - **Corollary**: communication follows session types

- We expect / would like to see:
  - **Progress**: Either system can take a step, or every process terminated
  - **Fidelity**: (Absent unguarded recursion), reduction in type environment should be reflected in processes
  - **Global progress**: (Absent unguarded recursion) all sessions should be able to reduce

Current Status

- Syntax, reduction rules, type system
- Type preservation proof
- Typechecker and small-step interpreter

Future Plans

- Stronger metatheory: global progress
- Switching between sessions in a **send** state
- Mainstream language implementation (e.g., via Scala API generation)

Conclusion

- Actor languages are popular tools for reliable distributed code
  - However, they are susceptible to communication errors
- Session types can rule out communication errors
  - However, they are difficult to apply to actors directly
- This work: first statically-checked application of session types to actors, where actors can take part in multiple sessions
  - Key idea: combine flow-sensitive effect typing and first-class event handlers

Thanks!

`@simon_jf@mastodon.scot`

Bonus Slides

$$\frac{\Gamma, x : A \mid S \triangleright M : B \triangleleft T}{\Gamma \vdash \lambda x.M : A \xrightarrow{S,T} B}$$

Abstraction:
Pre- and post-conditions recorded in function types

$$\frac{\Gamma \vdash V : A \xrightarrow{S,T} B \qquad \Gamma \vdash W : A}{\Gamma \mid S \triangleright V\ W : B \triangleleft T}$$

Application:
Recorded pre- and post-conditions must be compatible

$$\frac{\Gamma \vdash V : A}{\Gamma \mid S \triangleright \mathbf{return}\ V : A \triangleleft S}$$

Return:
Treat a value as a computation

$$\frac{\Gamma \mid S_1 \triangleright M : A \triangleleft S_2 \qquad \Gamma, x : A \mid S_2 \triangleright N : B \triangleleft S_3}{\Gamma \mid S_1 \triangleright \mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ N : B \triangleleft S_3}$$

Let:
Sequence effectful computations
The only evaluation context