Event-Driven Multiparty Session Actors

Simon Fowler
simon.fowler@glasgow.ac.uk
University of Glasgow
United Kingdom

Raymond Hu
r.hu@qmul.ac.uk
Queen Mary University of London
United Kingdom

Abstract
Actor languages such as Erlang and Elixir have emerged as popular tools for designing reliable, fault-tolerant distributed applications, but communication patterns used by actors are often informally specified. Multiparty session types (MP-STs) are a type discipline for communication protocols: if a program typechecks according to its session type, then it is guaranteed to fulfil its role in a communication protocol, but the unidirectional communication mechanism used by actors makes it difficult to apply session types to actor languages directly. By combining a flow-sensitive effect system with an event-driven programming model, we show the first statically-typed session type system for actors that can participate in multiple sessions.

1 Introduction
Actor languages and frameworks are a mainstay of reliable distributed software development: an actor is an addressable process that can spawn and send messages to other actors, and react to incoming messages. Actor languages support powerful idioms such as supervision hierarchies [1], which have helped companies such as Ericsson develop systems with "nine nines" reliability; Erlang is also used as the backbone of WhatsApp, with billions of users worldwide.

Alas, message passing introduces the threats of deadlocks and communication mismatches. Multiparty session types [9] encode communication patterns as types; successful type-checking ensures communication safety. To date, session typing for actors has either been checked dynamically [14], or has restricted each actor to a single session [7]. Restricting an actor to a single session is disadvantageous as it is often useful for an actor to have some common state (e.g., the stock of a warehouse) common to multiple sessions.

Problem statement. Multiparty session types can rule out communication errors, and actor languages support robust distributed programming. However, due to the mailbox-oriented communication model supported by actors, applying session types to actors is challenging.

How can we support static checking of multiparty session types in actors which can take part in multiple sessions?

In this extended abstract, we detail ongoing work showing how combining a flow-sensitive effect system (i.e., an effect system with pre- and post-conditions) with event-driven programming supports flexible actor programming with statically-checked MPSTs. Although we do not introduce any novel effects machinery, the relevance of this work to HOPE is how effect typing and event-driven programming can address a pressing open problem in the session types community.

2 Preliminaries
We begin by introducing some preliminary concepts.

(Multiparty) session types. Session types [8] are a type discipline that enforce conformance to communication protocols: if a program typechecks against its session type, then it is guaranteed to implement the protocol. Originally, session types described communication between two participants. Multiparty session types [9] allow communication patterns to be described between more than two participants: a global type describes the interactions between all participants, and can be projected to a local type for each participant.

Actor- and channel-based communication. A channel is a shared name or buffer that allows two or more processes to communicate. An actor is a process which can react to incoming messages; typically actor languages associate processes with a mailbox. Figure 1 (taken from [4], which provides a detailed formal comparison) shows the difference between the two models.

Whereas it is straightforward to give types to session channel endpoints, the unidirectional nature of mailboxes makes it difficult to apply session types to mailboxes directly. Therefore, session types are typically used to govern the actions that an actor performs, rather than the mailbox itself.

Event-driven programming. In the event-driven programming model, computation is triggered by an
event (for example, a mouse click, or an incoming message). Each event is handled by an event handler; after the handler has completed, the thread reverts to being idle.

Flow-sensitive effect systems. Standard Gifford-style type-and-effect systems [12] record the effects performed by an expression (e.g., writing to a reference cell or opening a file handle). These type-and-effect systems are typically set-based and do not reason about ordering of effects. In contrast, flow-sensitive [13] (also known as sequential [6]) effect systems reason about the ordering of effects, and their typing judgements typically express effects as pre- and post-conditions. Our functional formulation is reminiscent of Atkey’s parameterised monads [2].

Putting them together. Since it is difficult to attach session types to mailboxes, previous work has used session types to govern the communication actions performed by an actor, either using runtime monitoring [3, 14], or static typechecking for a single session using session types as pre- and post-conditions in a flow-sensitive effect system [7].

In short, our programming model combines the benefits of actor style programming (e.g., data locality) with statically-checked multiparty session types. Session typing is enforced by a type-and-effect system, and event-driven programming allows an actor to take part in more than one session.

3 Programming model by example

We introduce our model using the two-buyer protocol [9], intended as an abstraction of financial protocols. Two buyers (Buyer1 and Buyer2) interact with a Seller to buy a book: Buyer1 sends a title to Seller, who responds with a price. At this point, Buyer1 sends Buyer2 their share of the price. Buyer2 can then either accept the offer, sending their address to Seller and receiving a delivery date, or reject the offer. The global type for the protocol is described on the left; by projecting the global type, we obtain a local type for each role. The local type for Buyer2, B2, is shown on the right; & denotes branching and © denotes making a choice.

Although our core calculus is in the style of fine-grain call-by-value λ-calculus extended with constructs for event-based concurrency and session communication; we will describe the typing rules for each shortly.

Types include base types C, function types A → B (a function from A to B with session precondition S and post-condition T), and access point types AP( (p : S_i) ). Local session types S, T consist of input session types p &{i(A_i) . S_i} (ranged over by S) and output session
with the parameter name $x_{15}$.

Yoshida

$\rho$ out with the context of a session (in the context of role $T$). We model an actor as a triple $\langle T, \sigma, \rho \rangle$ since it will abort the current evaluation context. Suspending aborts the current evaluation context and adds the given handler to $\sigma$. We model asynchronous communication through the use of a session-level queue (which, due to message reordering rules, is isomorphic to individual role-level queues). Sending appends a message to the queue. If an actor is idle and has a stored event handler $s[p] \mapsto \text{handler} \ q \ \{ t_i(x_i) \mapsto M_j \}_{i,j}$ and the head of the queue contains a message $\langle q, p, t_i(V_j) \rangle$ (where $j \in I$), then the thread state will become $(M_j(V_j/x_j))^{s[p]}$.

**Metatheory.** We have proved a type preservation theorem using the generalised approach introduced by Scalas and Yoshida [15]. We conjecture that the event-driven nature of the system will also permit a global progress result like that of Viering et al. [16], although this is a work-in-progress.

5 Conclusion

We have shown how the combination of a flow-sensitive effect system and event-driven programming allows static typing of actors that participate in multiple sessions. We have also implemented a typechecker and small-step interpreter for the calculus; we further plan to build on the code generation approach introduced by Hu and Yoshida [10] to allow the programming model to be used in mainstream programming languages such as Scala. Our next steps are to prove a global progress theorem and to investigate how to switch between sessions while maintaining progress guarantees.

Acknowledgments

Thanks to Matthew Alan Le Brun and Alceste Scalas for discussion of the theory of asynchronous generalised multi-party session types. Thanks also to the HOPE‘23 reviewers for their valuable comments; hopefully the abstract is more accessible as a result. Fowler was funded by EPSRC grant EP/T014628/1 (STARDUST).

References


A Full two-buyer example

Local types.

\[
B_1 \triangleq \text{Seller} \oplus \text{title}((\text{String}), \text{Seller} \oplus \text{quote}((\text{String}), \text{Buyer}_2 \oplus \text{share}((\text{Int}), \text{end}) \\
B_2 \triangleq \text{Buyer}_1 \oplus \text{share}((\text{Int}), \text{Seller} @\{}
\begin{array}{l}
\text{address}((\text{String}), \text{Seller} \oplus \text{date}((\text{Date}), \text{end}) \\
\text{quit}((\text{Unit}), \text{end})
\end{array}
\text{Buyer}_2 &\{}
\begin{array}{l}
\text{address}((\text{String}), \text{Buyer}_1 \oplus \text{quote}((\text{Int}), \text{Buyer}_2 \oplus\{}
\begin{array}{l}
\text{address}((\text{String}), \text{Buyer}_1 \oplus \text{date}((\text{Date}), \text{end}) \\
\text{quit}((\text{Unit}), \text{end})
\end{array}
\text{S} \triangleq \text{Buyer}_1 \oplus \text{title}((\text{String}), \text{Buyer}_1 \oplus \text{quote}((\text{Int}), \text{Buyer}_2 &\{}
\begin{array}{l}
\text{address}((\text{String}), \text{Buyer}_2 \oplus \text{date}((\text{Date}), \text{end}) \\
\text{quit}((\text{Unit}), \text{end})
\end{array}
\]

Main function.

\[
\text{main} \triangleq \begin{array}{l}
\text{let } ap \leftarrow \text{newAP} \text{ (Seller-S, Buyer1-B1, Buyer2-B2) in}
\text{spawn (seller}(ap) \{}; \text{spawnBuyers}(ap, "Types and Programming Languages"); \text{spawnBuyers}(ap, "Compiling with Continuations")
\end{array}
\]

\text{spawnBuyers}(ap, title) \triangleq \text{spawn buyer1}(ap, title); \text{spawn buyer2}(ap)

Seller.

\[
\text{seller}(ap) \triangleq \text{rec install(\_).}
\begin{array}{l}
\text{register ap Seller \{}
\begin{array}{l}
\text{install}();
\text{suspend titleHandler}
\end{array}
\text{titleHandler} \triangleq \text{handler Buyer1 \{}
\begin{array}{l}
\text{title(x) \rightarrow Buyer1!quote(lookupPrice(x));}
\text{suspend decisionHandler}
\end{array}
\text{handler Buyer2 \{}
\begin{array}{l}
\text{address(addr) \rightarrow Buyer2!date(shippingDate(addr))}
\text{quit()} \rightarrow \text{return ()}
\end{array}
\text{end}
\end{array}
\end{array}
\]

Here, shippingDate is left abstract and calculates a shipping date given an address.

Buyer 1.

\[
\text{buyer1}(ap, title) \triangleq \text{register ap Buyer1 \{}
\begin{array}{l}
\text{Seller!title(title);}
\text{suspend quoteHandler}
\end{array}
\text{quoteHandler} \triangleq \text{handler Buyer \{}
\begin{array}{l}
\text{quote(amount) \rightarrow Buyer2!share(amount/2)}
\end{array}
\text{end}
\]

Buyer 2.

\[
\text{buyer2}(ap) \triangleq \text{register ap Buyer2 (suspend shareHandler)}
\]

\text{shareHandler} \triangleq \text{handler Buyer \{}
\begin{array}{l}
\text{share(amount) \rightarrow}
\text{if (amount > 100) then}
\text{Seller!quit()}
\text{else}
\text{Seller!address("18 Lilybank Gardens");}
\text{suspend dateHandler}
\end{array}
\text{dateHandler} \triangleq \text{handler Seller \{}
\begin{array}{l}
\text{date(date) \rightarrow log(date)}
\end{array}
\]

Here, log is left abstract, and logs the received date.
B  Full typing rules

Value typing

\[ \Gamma \vdash V : A \]

\begin{align*}
\text{TV-VAR} & \quad \text{TV-LAM} & \quad \text{TV-REC} & \quad \text{TV-CONST} \\
\Gamma \vdash x : A & \in \Gamma \quad \Gamma ; x : A \mid S \vdash M : B \leadsto T & \quad \Gamma ; x : A, f : A \xrightarrow{S_T} B \mid S \vdash M : A \xrightarrow{S_T} B \leadsto T & \quad c \text{ has base type C} \\
\Gamma \vdash \lambda x . M : A \xrightarrow{S_T} B & \quad \Gamma \vdash \text{rec } f(x) . M : A \xrightarrow{S_T} B & \quad \Gamma \vdash c : C
\end{align*}

\text{TV-HANDLER}

\[ (\Gamma ; x : A_1 \mid S_1 \vdash M_1 : 1 \leadsto \text{end})_i \]

\[ \Gamma \vdash \text{handler } p \{ t_1(x_i) \mapsto M_1 \}_i : \text{Handler}(p \& \{ t_1(A_1), S_1 \}_i) \]

Computation typing

\begin{align*}
\text{T-RETURN} & \quad \text{T-LET} & \quad \text{T-APP} & \quad \text{T-SPAWN} & \quad \text{T-SEND} & \quad \text{T-REGISTER} \\
\Gamma \mid S \rightarrow \text{return } V : A & \quad \Gamma \mid S \rightarrow \text{let } x \leftarrow M \text{ in } N : B \leadsto S_1 & \quad \Gamma \mid S \rightarrow V : A \xrightarrow{S_T} B \mid \Gamma \vdash W : A & \quad \Gamma \mid S \rightarrow \text{spawn } M : 1 \leadsto S & \quad \Gamma \mid p \uplus \{ t_i(A_i), S_i \}_{i \in I} \rightarrow p! t_j(V) : 1 \leadsto S_j & \quad \Gamma \mid S \rightarrow \text{register } V p_j : M : 1 \leadsto S
\end{align*}

\begin{align*}
\text{T-IF} & \quad \text{T-SUSPEND} & \quad \text{T-NEWAP} & \quad \text{T-REGISTER} \\
\Gamma \mid V : \text{Bool} & \quad \Gamma \mid S' \rightarrow \text{suspend } V : A \leadsto T & \quad \varphi \text{ is a safety property } \varphi((p_t : T_i)_i) & \quad j \in I \quad \Gamma \mid V : \text{AP}((p_t : T_i)_i) & \quad \Gamma \mid T_j \rightarrow M : 1 \leadsto \text{end} & \quad \Gamma \mid S \rightarrow \text{register } V p_j : M : 1 \leadsto S
\end{align*}

\[ \Gamma \mid S \rightarrow \text{if } V \text{ then } M \text{ else } N : A \leadsto S_1 \]

\[ \Gamma \mid S \rightarrow \text{suspend } V : A \leadsto T \]

\[ \varphi((p_t : T_i)_i) \]

\[ \varphi((p_t : T_i)_i) \rightarrow \text{AP}((p_t : T_i)_i) \rightarrow S \]