First-class Distributed Session Types

Simon Fowler
University of Edinburgh
simon.fowler@ed.ac.uk

1 Introduction

Communication-centric programming languages such as Erlang, Go, and Pony put communication and concurrency at the centre of their design, providing lightweight processes which co-ordinate through the use of message-passing, making applications easier to structure and reason about than shared-memory systems.

Communication follows *protocols*, either for external communication (such as SMTP [14]) or for internal communication between different components of a system. How do we document the inherent communication patterns in an application, and how do we ensure that applications conform to these protocols?

Session types [6, 7] encode communication patterns as types. We describe the design and implementation of a distributed extension to the web-based, session-typed functional language Links, allowing the creation of multi-user web applications which can communicate using first-class session-typed channels. We outline ongoing work adapting the work of Mostrous and Vasconcelos [12] to the setting of functional languages, in order to handle the case where participants go offline during the course of a session. Distributed Session Links is available at http://www.github.com/links-lang/links.

2 Background

Go [1] provides simply-typed channels such as chan(int). Session types allow more expressive types: for example, we can describe a "calculator server" (example originally due to Gay and Hole [5]), allowing the choice between addition and negation operations:

$$\begin{aligned}\text{CalcServer} = \quad \&\{ \quad & \text{add} : ?\text{Int}.?\text{Int}.!\text{Int}.\text{CalcServer}; \\ & \text{neg} : ?\text{Int}.!\text{Int}.\text{CalcServer} \quad \}\end{aligned}$$

Here, the & operator *offers* a choice between the add and neg operations. The add operation receives two integers and sends an integer (where receiving is denoted by ? and sending is denoted by !), whereas the neg operation receives a single integer before sending an integer. The client would have the *dual* session type:

$$\begin{aligned}\text{CalcClient} = \quad \oplus\{ \quad & \text{add} : !\text{Int}.!\text{Int}.?\text{Int}.\text{CalcClient}; \\ & \text{neg} : !\text{Int}.?\text{Int}.\text{CalcClient} \quad \}\end{aligned}$$

Note here that where the server sends a message, the client receives a message, and where the server offers a choice, the client makes a selection (denoted by ⊕). Session types guarantee that communication follows the protocol described by the session type, preventing communication mismatches and deadlocks along a single channel.

On the Challenge of Linearity The main technical hurdle in implementing session types is that of *linearity*. Intuitively, we wish to ensure that each "step" of a session channel is only used once. Consider the following function:

```
sig f : (!Int.?Int.end) ~> Int
fun f(s){
  var t = send (5, s);  var s = send (5, s);
  var (x, s) = receive s;  x
}
```

Here, function `f` takes a session type `s` of type `!Int.?Int.end`: that is, a channel over wish we wish to send an integer and then receive an integer. However, naïvely implemented, we could simply re-use the first endpoint, sending an integer twice, and losing all guarantees given by the session type! Linearity ensures that each endpoint is used *exactly once*, ensuring that endpoints are not reused, and that all communication actions are completed.

Embeddings in mainstream languages such as Haskell [10, 13, 15] and Scala [16] check linearity at runtime or through advanced type system features, but can introduce embedding artefacts and can have complex error messages. Another approach is to add session types as first-class constructs. Recent work extends the Links [3] web-based functional language with first-class session types [11].

3 Links: From Multithreaded to Distributed

Links provides supports both simply-typed actor-style processes and session-typed channels, which imply a degree of location transparency: for example, we should be able to send a message over a channel regardless of the location of the peer endpoint. In spite of this, prior to this work each concurrency runtime remained stubbornly independent, introducing the "barriers" shown in Figure 1.

This work liberates the abstractions by breaking the barriers between the concurrency runtimes.

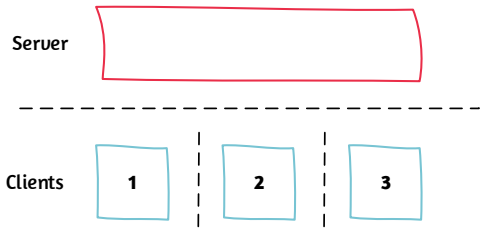

Figure 1. "Barriers" between concurrency runtimes

A Simple Example Consider a web application which comprises two types of client: a "pinger" which sends a `Ping` message and receives a `Pong` message, and a "ponger" which receives a `Ping` message and sends a `Pong` message.

We begin by describing the `PingPong` session type, describing a channel which receives a `Ping`, sends a `Pong`, and then finishes.

```
typename PingPong = ?(Ping) . !(Pong) . End;
```

We firstly create an *access point*, `ap`, which acts as a "matchmaking service" for processes to establish sessions. We add two routes, `/pinger` and `/ponger`, which take a callback function passing the URL and a *location*, returning the page to be displayed. The calls to `serveWebsockets` and `servePages` start the webserver.

```
fun main() {
  var ap = new();
  addRoute("/pinger", fun(_, loc) { Pinger.page(loc, ap) });
  addRoute("/ponger", fun(_, loc) { Ponger.page(loc, ap) });
  serveWebsockets(); servePages()
}
```

Both the `Pinger` and `Ponger` provide a function `page` which takes the location and the access point as arguments, and produces a page (asssuming a function `simplePage` returning some HTML). The `spawnAt` construct spawns a function at a given location, in this case on the client. The `Pinger` *requests* from the access point, obtaining an endpoint which is the dual of the `PingPong` type. The process then sends a `Ping`, logs this to the browser console, then receives and logs a `Pong`. The `Ponger` performs the dual actions.

```
module Pinger {
  sig page : (Location, AP(PingPong)) ~> Page
  fun page(loc, ap) {
    var pid = spawnAt( loc,
      { var ch = request(ap);
        var ch = send(Ping, ch); print("Sent Ping!");
        var _ = receive(ch); print("Received Pong!") });
    simplePage()
  } }
module Ponger {
  sig page : (Location, AP(PingPong)) ~> Page
  fun page(loc, ap) {
    var pid = spawnAt( loc,
      { var ch = accept(ap);
        var (_, ch) = receive(ch); print("Received Ping!");
        var _ = send(Pong, ch); print("Sent Pong!") });
    simplePage()
  } }
```

Links handles all session type checking, session establishment, data serialisation and deserialisation, and session teardown transparently. A more full-featured example, that of a distributed web-based chat server, can be found at https://github.com/links-lang/links/blob/master/examples/distribution/chatserver/.

Breaking the Barriers The main concepts in the implementation are as follows:

First-Class Locations and Closure Serialisation We make the notion of a *location* first-class, explicitly allowing processes to be spawned on a given client. To do so, we closure-convert an application, delivering an initial state containing the functions to be run on the client.

Generalised Process IDs and Channel IDs We generalise process IDs to include locations. Channel endpoints consist of a pair of endpoint IDs; each runtime tracks local endpoints. Communication with an external process or peer channel endpoint results in an external request.

Websockets Whereas Links previously used AJAX requests to encode remote procedure calls [3, 4], Distributed Links supports bidirectional communication through the use of websockets.

Server-side Routing Web clients cannot connect to each other directly, so it is necessary to have some server-side routing. The server keeps track of the locations of each endpoint and process, which also involves inspecting messages for process and channel IDs which were created on a client.

Distributed Delegation *Delegation* allows session endpoints to be sent along other session endpoints. Alas, in the presence of asynchrony *and* distribution, this proves to be challenging (see Hu et al. [8]). We have devised a distributed delegation algorithm which works in the more restricted web-based setting, by synchronising with a client to reobtain "lost messages", and sequentialising delegation requests to ensure that a "carrier channel" is not delegated before a previous delegation is complete.

4 Affine Sessions with Exceptional Syntax

When writing web applications with session types, we *cannot expect linearity*: users may, of course, simply close their browsers before a session has completed! As a result, processes simply become stuck waiting for a message which will never arrive.

Mostrous and Vasconcelos [12] describe a synchronous process calculus which can relax the requirement of *linearity* to that of *affinity* (that each endpoint must be used at most once), which precisely captures our scenario. The authors provide an exception handling construct which attempts to perform a communication action ρ along a channel a with session type S. If the communication action fails (for example, if the partner endpoint is unavailable), then process P (typeable *without* a) is evaluated.

$$\frac{\Gamma, a : S \vdash \rho \qquad \Gamma \vdash P \qquad \mathsf{subject}(\rho) = a}{\Gamma, a : S \vdash \mathsf{do}\ \rho\ \mathsf{catch}\ P}$$

Alas, this does not adapt straightforwardly to our setting of an asynchronous concurrent λ-calculus. Consider the following:

```
sig recvAndAdd : (?Int.end, ?Int.end) ~> Int
fun recvAndAdd(s, t) {
  try {
    var (x, s) = receive s; var (y, t) = receive t;
    x + y
  } catch { (-1) }
}
```

Here, we have a program which attempts to receive two integers along two different channels, returning their sum should both receives succeed, or returning (-1) should either operation fail. This example is not typeable using the previous construct–either s or t would have to be used within the catch block!

Instead, we propose a construct inspired by Benton & Kennedy's Exceptional Syntax [2], with the following typing rule:

$$\frac{\Gamma_1 \vdash M : A \qquad \Gamma_2, x : A \vdash N : B \qquad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash \mathsf{try}\ M\ \mathsf{as}\ x\ \mathsf{in}\ N\ \mathsf{otherwise}\ N' : B}$$

Here, M is a term which contains a potentially-failing operation, such as sending to a channel whose partner endpoint is unavailable. Should M evaluate correctly, it is bound to x in N. Otherwise, N' is evaluated. This formulation allows us to type our program:

```
fun recvAndAdd(s, t) {
  try {
    var (x, s) = receive s;
    var (y, t) = receive t; (x, y)
  } as (x, y) in { (x + y) } otherwise { (-1) }
}
```

We have formalised the construct as an extension to an asynchronous variant of the GV calculus [9]. The key idea is to stratify evaluation contexts into pure contexts and exception handling contexts, and to inspect the free channel variables of the possibly-failing term upon a communication failure, disabling all affected channels. Implementing this calculus in Links is the subject of ongoing work.

5 Conclusion

We have described an extension to the Links programming language to support the use of session types in the web-based distributed setting. We have outlined the design and implementation, and have described ongoing work on the design of construct able to handle participants going offline during a session. Future work includes refining and implementing the core calculus for affine sessions, as well as adding the ability to interact with other Links servers in addition to web clients.

References

[1] 2017. The Go Programming Language. https://golang.org/. (2017).

[2] Nick Benton and Andrew Kennedy. 2001. Exceptional syntax. *Journal of Functional Programming* 11 (2001), 395–410. https://doi.org/10.1017/s0956796801004099

[3] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects (FMCO '06)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.), Vol. 4709. Springer Berlin Heidelberg, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

[4] Ezra E. K. Cooper and Philip Wadler. 2009. The RPC calculus. In *PPDP*. ACM, 231–242.

[5] Simon Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2 (22 Nov. 2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

[6] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Lecture Notes in Computer Science, Vol. 715. Springer Berlin Heidelberg, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[7] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Lecture Notes in Computer Science, Vol. 1381. Springer Berlin Heidelberg, Berlin/Heidelberg, Chapter 9, 122–138. https://doi.org/10.1007/bfb0053567

[8] Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-Based Distributed Programming in Java. In *ECOOP*, Jan Vitek (Ed.). Lecture Notes in Computer Science, Vol. 5142. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter 22, 516–541. https://doi.org/10.1007/978-3-540-70592-5_22

[9] Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *ESOP*. Springer, 560–584.

[10] Sam Lindley and J. Garrett Morris. 2016. Embedding session types in Haskell. In *Proceedings of the 9th International Symposium on Haskell*. ACM, 133–145.

[11] Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: From Theory to Tools*, Simon Gay and Antonio Ravara (Eds.). River Publishers.

[12] Dimitris Mostrous and Vasco T. Vasconcelos. 2014. Affine Sessions. In *Coordination Models and Languages*, Eva Kühn and Rosario Pugliese (Eds.). Springer Berlin Heidelberg, 115–130. https://doi.org/10.1007/978-3-662-43376-8_8

[13] Matthias Neubauer and Peter Thiemann. 2004. An Implementation of Session Types. In *Practical Aspects of Declarative Languages*, Bharat Jayaraman (Ed.). Lecture Notes in Computer Science, Vol. 3057. Springer Berlin Heidelberg, 56–70. https://doi.org/10.1007/978-3-540-24836-1_5

[14] J. Postel. 1982. *Simple Mail Transfer Protocol*. Technical Report 821. RFC Editor, Fremont, CA, USA. http://www.rfc-editor.org/rfc/rfc821.txt

[15] Riccardo Pucella and Jesse A. Tov. 2008. Haskell Session Types with (Almost) No Class. *SIGPLAN Not.* 44, 2 (Sept. 2008), 25–36. https://doi.org/10.1145/1543134.1411290

[16] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.