Dynamic Session Types

Student Number: 170027195

June 16, 2018

Contents

1 Literature Review

1.1 Session Types

When systems attempt to communicate over a protocol, the constraints of "acceptable" communication must be defined before and during transmission. The rubric to guarantee that protocol communication has fit within these boundaries, is however, less clear. If errors in transmission are to occur, future communication may become infeasible, and at worst, security issues could be introduced.

Honda derived session types[1][2][3] as a strategy to mitigate potential issues that stem from the inability to properly check that an implementation of the protocol correctly matches the standard. In the 1994 paper[2] they describe the need for an "abstraction methodology"to describe interactions between "multiple, possibly distributed, parallel processes", going on to state "Such organization in turn may well be based on a type discipline, which ensures well-typed programs to behave in a principled fashion". At its core, the session types proposal is for a language or method of describing the expected protocol communication, with a mechanism to check protocol implementations against the type definition, ensuring their correctness. Session types take influence from π-calculus[4], a process calculus used to model concurrent systems, with the 1994 paper[2] showing the authors adaption of π-calculus model of "synchronous communication of values".

After the protocol representation has been described, often using the Scribble formal description language[5], a checking process can occur to ensure that the implementation of the protocol matches its structure. The checking process can occur using one of three mechanisms, either statically, where the validity of the implementation is checked at compile time, dynamically at run time, or a hybrid approach is used, a combination of compile time assurance with a layer of run time checking also enforced. Such a hybrid approach is described in Hu and Roshida, 2016[6] a static checker is used to validate the "behavioural" aspects of the system, generating finite state machines and restricting operations that can be performed adhere to a defined state. They then describe a "light" dynamically checked solution to ensure that defined channels of communication are used, and then are not re-used without throwing an error.

1.2 The Actor Model

The actor model introduces a model for the control of parallel computation to avoid potential errors, such as deadlock or race conditions. The actor model places a number of rules upon what an actor can do, an actor has 3 valid "actions"; to create another actor, to send a message to another actor or to instruct itself on how to deal with the next message it receives.

Each actor is assigned with some concept of an "address", this provides a mechanism for an actor to communicate with other actors within the system, passing messages between each other as needed. When messages are passed from one actor to another, they are stored in the actors individual task queue; these tasks are then executed one by one, with an actor polling the queue for new tasks after a single unit of work has been completed. Actors also have some form of private state, this allows for an actor to be aware of

information relating to the current status of its execution and the execution of tasks prior to the task is is currently executing. Through these three components, the address, the individual task queue and the private state, a sense of the decoupled nature of the actor model can be realised. The goal is to fully abstract and decouple actors from each other, with communication and synchronisation between actors only occurring via the message passing framework.

There are a variety of implementations of the actor model, some research-focused languages are designed entirely entire around the concept, such as Rosette[7]. Other implementations that are more popular amongst the wider general purpose programming community are Erlang's concurrent and distributed computation utilities, and Akka, which provides an actor model framework for languages that run on the Java Virtual Machine, such as Scala, Java and Kotlin. Erlang is a functional programming language which has built in message passing capabilities for communication between processes. The `!` operator is used to send a message to the identified process, an example of a program which sends the word "Ping" to an actor, which then returns "Pong" a number of times is listed in the Erlang documentation[8].

Listing 1: Ping Pong in concurrent Erlang[8]

```
1  -module(tut15).
2
3  -export([start/0, ping/2, pong/0]).
4
5  ping(0, Pong_PID) ->
6          Pong_PID ! finished,
7          io:format(''ping finished~n'', []);
8
9  ping(N, Pong_PID) ->
10         Pong_PID ! {ping, self()},
11         receive
12                 pong ->
13                         io:format(''Ping received pong~n'', [])
14         end,
15         ping(N - 1, Pong_PID).
16
17 pong() ->
18         receive
19                 finished ->
20                         io:format(''Pong finished~n'', []);
21                 {ping, Ping_PID} ->
22                         io:format(''Pong received ping~n'', []),
```

```
23                          Ping_PID ! pong,
24                          pong()
25          end.
26
27  start() ->
28          Pong_PID = spawn(tut15, pong, []),
29          spawn(tut15, ping, [3, Pong_PID]).
```

1.3 Static Session Type Implementations

Static implementations of session types, as expressed in section 1.1 perform the checking of the validity of the protocol implementation at compile time. There are multiple real world implementations for a host of languages that implement such a system in varying forms, from Java through to Haskell[9].

Some statically typed implementations, such as SessionJ[10] for Java implement extensions to the language they are built for to provide this functionality. SessionJ implements a `protocol` keyword and subsequent domain specific language for expressing the contents of a protocol specification.

1.4 Dynamic Session Types Implementations

1.5 Security Problems Regarding Protocols

References

[1] Kohei Honda. "Types for dyadic interaction". In: *International Conference on Concurrency Theory*. Springer. 1993, pp. 509–523.

[2] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. "An interaction-based language and its typing system". In: *International Conference on Parallel Architectures and Languages Europe*. Springer. 1994, pp. 398–413.

[3] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. "Language primitives and type discipline for structured communication-based programming". In: *European Symposium on Programming*. Springer. 1998, pp. 122–138.

[4] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[5] Kohei Honda et al. "Scribbling interactions with a formal foundation". In: *International Conference on Distributed Computing and Internet Technology*. Springer. 2011, pp. 55–75.

[6] Raymond Hu and Nobuko Yoshida. "Hybrid session verification through endpoint API generation". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2016, pp. 401–418.

[7] C. Tomlinson et al. "Rosette: An Object-oriented Concurrent Systems Architecture". In: *SIGPLAN Not.* 24.4 (Sept. 1988), pp. 91–93. ISSN: 0362-1340. DOI: 10.1145/67387.67410. URL: http://doi.acm.org/10.1145/67387.67410.

[8] Erlang Documentation. *Concurrent Programming in Erlang*. 2018. URL: http://erlang.org/doc/getting_started/conc_prog.html (visited on 06/13/2018).

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

[10] Raymond Hu et al. "Type-safe Eventful Sessions in Java". In: *Proceedings of the 24th European Conference on Object-oriented Programming*. ECOOP'10. Maribor, Slovenia: Springer-Verlag, 2010, pp. 329–353. ISBN: 3-642-14106-4, 978-3-642-14106-5. URL: http://dl.acm.org/citation.cfm?id=1883978.1884001.