Dynamic Session Types

Student Number: 170027195

June 13, 2018

Contents

1 Literature Review

1.1 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[2]. 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[1].

Listing 1: Ping Pong in concurrent Erlang[1]

```
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]).
```

1.2 Implementations of Session Types in Object Orientated Languages

References

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

[2] 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.