DynoSesh: Dynamic Session Types

Student Number: 170027195

July 24, 2018

Contents

List of Figures

Listings

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] the need for an "abstraction methodology" to describe interactions between "multiple, possibly distributed, parallel processes" is outlined, 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 a usable protocol representation has been outlined, 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, a combination of compile time assurance with a layer of run time checking is employed. Such a hybrid approach is described in Hu and Roshida, 2016[6] during which a static checker is used to validate the "behavioural" aspects of the system, generating finite state machines and restricting the operations that can be performed to those which adhere to one of the defined states. They then describe a "light" dynamically checked solution to ensure that defined channels of communication are used, and then are not re-used at a later date without the error in doing so violating the protocol definition.

Modern research on session types considers "multiparty" systems, while early research largely considered a protocol implementation that exhibited communications between two parties, multiparty communication can have any numbers of participants in protocol communication. A 2008 paper by Honda et al.[7] gives an example of protocol communication which is extremely difficult to represent in a binary session, giving need for a solution which considers multiparty communication. The consideration is for a system in which users can purchase expensive books from a seller, two buyers wish to purchase a single book together, the first buyer sends a book title to the seller, who responds to both the first and second buyer with the quote

for the price of the book, the first buyer then communicates to the second buyer how much of the price they will pay, the second buyer then finally accepts the quote if they wish to proceed. The authors describe this as "extremely awkward (if logically possible) to decompose this scenario into three binary sessions" and express the need for the ability to express this as a single session, validated against a protocol implementation. The paper provides an expansion of the initial session types language found in[2, 3] which incorporates the ability to consider protocols which express the communication between multiple parties over a protocol.

1.1.1 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[8].

Some statically typed implementations, such as SessionJ[9] 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, an example of this is show in listing 1. The communication of the structure of a protocol may not be possible, or elegant, without creating such an extension, in the Hu et al. paper[9] an integration of the language extension into the wider Java compilation process is implemented. They describe three layers of compilation and execution, with the first layer being the "SessionJ source code", the Java code with the protocol extensions implemented, this then is translated into Java by the SessionJ compiler, before running on the JVM with SessionJ run time libraries to ensure that the sending an receiving of messages over the protocol is implemented correctly.

Listing 1: An implementation of a protocol to place an order by Hu et al. to showcase the capabilities of SessionJ[9]

```
1  protocol placeOrder {
2          begin. // Commence session.
3          ![       // Can iterate:
4                   !<String>.       // send String
5                   ?(Double)        // receive Double
6          ]*.
7          !{       // Select one of:
8                   ACCEPT: !<Address>.?(Date),
9                   REJECT:
10         }
11 }
```

Other approaches to the implementation of statically checked session types within a language do not extend or modify the language to achieve this functionality. One example, for the Rust programming language[10] uses chained type parameters to achieve their goal, they provide an example of the implementation of an ATM shown in listing 2, which can take withdrawals and deposits. This displays the syntax with which the session type framework is implemented, through the chaining of a number of calls to types, with `Recv<>` being to receive a value, and `Send<>` to send a value over the protocol. Any larger implementation that the ATM example would begin to have extremely long and convoluted type definitions to express the structure of the protocol, effectively becoming a domain specific language for the expression of a protocol's operation within another language. Due to these complexities, such an approach, particularly in the case of statically checked session types which has some form of compilation step to check the validity of the implementation at compile time, does not seem to be favoured.

Listing 2: An implementation of an ATM using session types in Rust[10]

```
1         type Atm = Recv<Id, Choose<Rec<AtmInner>,Eps>>;
2         type AtmInner = Offer<AtmDeposit, Offer<AtmWithdraw, Quit>>>;
3 where:
4         type Id = String;
5         type Quit = Eps;
6         type AtmDeposit = Recv<u64, Send<u64, Var<Z>>>;
7         type AtmWithdraw = Recv<u64, Choose<Var<Z>,Var<Z>>>;
```

1.1.2 Dynamic Session Type Implementations

There are also a wide number of dynamically checked session type implementations for a host of languages and technologies, one such tool is Session Python for the Python programming language[11]. The Neykova et al. paper stresses the need for static implementations to extend the language they are built for, "Static session type checking in these mainstream languages, however, requires support in the form of the language extensions and pre-compiler processing to be tractable". The implementation instead uses Scribble[5] to express the structure of protocol communication, this protocol structure is then enforced at run time.

In comparison to statically checked session types, where the correctness of an implementation is proved at compile time, runtime checking of session types require some level of coordination between actors using session types to check the status of their protocol communication amongst each other. In a distributed context, the global protocol communication standard shared amongst actors must be disseminated to a local context per each actor and enforced at that level locally. Without this, a single actor would be aware of the success of protocol communication locally, but would have no knowledge of the success of

protocol communication between other actors. The Session Python runtime provides each defined role in the protocol with its own local success parameters "Scribble (i.e. syntactic) local protocol specifications for each participant (role) defined in the protocols"[11], this ensures that there is global compliance with the protocol, even when actors are only aware of the constraints on their local context.

1.2 The Actor Model

The actor model introduces a strategy 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[12]. Other implementations that are more popular amongst the wider general purpose programming community are Erlang's concurrent and distributed computation utilities, and Akka[13], 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[14] as displayed in listing 3.

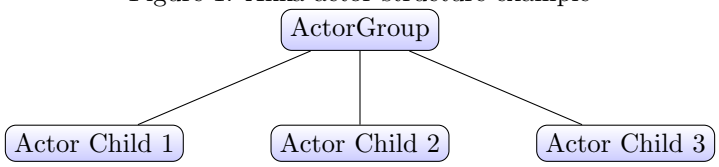
Listing 3: Ping Pong in concurrent Erlang[14]

```
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]).
```

Akka's implementation of the actor model provides a similar construct, actors exist within a wider network and pass messages to each other to execution methods. However, unlike Erlang, it provides this as library functionality on top of other general purpose programming languages that run on the Java Virtual Machine. Akka allows developers to create a tree hierarchy of actors, with individual actors organised into groups, as shown in figure 1.2. This hierarchy is used to enforce concurrency amongst and between groups, with actors inside a group running concurrently as well as the groups themselves. The hierarchical structure of the actor grouping also helps to isolate errors, if a failure of an actor occurs within a group, due to the isolated nature of the actor model this can be dealt with at the group level without requiring a full system-wide recovery process.

Figure 1: Akka actor structure example



1.3 Security Problems Regarding Protocols

There is a wide ranging literature on the security of protocols and their usage, covering strategies to guarantee secure communication and vulnerabilities that exist within current protocols. One such protocol is Kerberos[15] which provides the ability for actors to communicate securely over an unsecured network. The protocol is complicated, with many encrypted messages sent back and forth from client to client to prove the identity of an actor to prevent impersonation. An unsound implementation of the protocol could mean that vulnerabilities exist within communications between clients that were assumed to be secure, exposing the users unwittingly to harmful actors.

Faulty implementations of protocols have existed in the wild and potentially have left users of the implementation at risk of harm. Apple's implementation of the secure sockets layer[16] (SSL), protocol in iOS 6.0 was vulnerable to man in the middle attacks[17] which would allow for a hostile actor to "eavesdrop" on assumed secure communication between two parties. The code section which introduce the bug, shown in listing 4 occurred when the second `goto fail;` command was automatically executed on every time the `update()` function returned true, jumping over a critical final section of the algorithm. Such a bug is hard to identify, even after code review, yet has a wide effect on the sense of trust that users have in their devices to facilitate secure communication, the importance on correct and secure implementations of protocols is paramount.

Listing 4: The vulenarble code section in iOS 6.0's SSL implementation[17]

```
1 if ((err = SSLHashSHA1.update(
2         &hashCtx, &signedParams)) != 0)
3         goto fail;
4         goto fail;
```

2 Implementation

2.1 Protocol Representation

One of the key tasks with regards to the implementation of DynoSesh was producing a coherent strategy to represent the structure of protocols within the system. While approaches for this problem exist, none were fit for application in my project. The academic discourse around session types, as explained earlier in the literature review, largely concerns statically checked implementations, which are either structured via an external domain specific language or an extension to the implementation language.

One considered approach was a "chained types" implementation, this would have provided a set of types in Java to construct a larger session. An example of this would have been a "decision" type, which took two type parameters, with the signature `Decision<S,T>` where supplying S or T would have forced the protocol to take a different path depending on the input. However as displayed in a paper which uses this approach earlier[10], any reasonably complicated protocol implementation using this appproach has a unruly type signature which is hard to parse and harder to create. As the aim of the project is to create an easily usable "plug and play" library to enhance and improve safety and security in Java programs, it is important to avoid obnubilating protocol structure from a reader.

During the development process, the settled upon protocol representation strategy resembled an implementation of Finite State Machines. Finite state machines are a form of directed graph, a collection of nodes connected to each other with edges, representing "valid" states a system can have and what legal moves can be made between these states depending on certain conditions. In the protocol graph, a "state" is a node which contains a type, these nodes are then connected with edges which have a guard specifying which actor can traverse along them. An example of a graph which allows to actors to say hello to each other before terminating the communication would have three nodes. The first node, would have a `null` payload and an edge containing a reference to actor 0 to another node which contained `Hello.class` finally this node would have an edge containing a reference to actor 1 to a final node that also had the payload `Hello.class`.

2.2 Graph Traversal

2.3 Domain Specific Language

Users having the ability to express the structure of complex protocols easily is key to the adoption of the library by a wider user-base. It is paramount that this is made as simple as it possibly can, to improve

both usability when creating the protocol structure and comprehension when checking it at a later date. As expressed in the literature review, one such standard approach is the Scribble[5] domain description language, where users create protocols and then use tools to compile the contents to an implementation usable with their language of choice. In a library sense, however, this is imperfect, it requires an extra layer of knowledge from users outside of the normal language paradigm, it increases the learning curve, potentially putting off new users, and requires some form of parsers and compilation process which could prove difficult to implement in the time constraints of the project.

Within the domain specific language discourse there is a distinction made between so called "external" and "internal" domain specific languages[18]. External implementations, such as Scribble, are written in a language outside of the implementation language, often within a completely separate file, they are then parsed and comprehended by the runtime that has been implemented to execute them. Internal domain specific languages, however, are written within their implementation language, complying validly with the grammar. A valid expression written using an internal domain specific language will also be valid within the wider implementation language that was used to build it. Fowler provides two examples of this in his book, showing the difference between "standard" APIs, shown in listing 5, and an implementation of an internal domain specific language that utilises method chaining displayed in listing 6, both providing the same result.

Listing 5: Object construction in a normal API[18]

```
1  Processor p = new Processor(2, 2500, Processor.Type.i386);
2  Disk d1 = new Disk(150, Disk.UNKNOWN_SPEED, null);
3  Disk d2 = new Disk(75, 7200, Disk.Interface.SATA);
4  return new Computer(p, d1, d2);
```

Listing 6: Object construction in a method chained internal Domain Specific Language[18]

```
1  computer()
2          .processor()
3                  .cores(2)
4                  .speed(2500)
5                  .i386()
6          .disk()
7                  .size(150)
8          .disk()
9                  .size(75)
10                 .speed(7200)
11                 .sata()
12         .end();
```

While the latter of the two listings does not look like "regular" Java code, or follow normal recommended standards surrounding line length, it still compiles. If a wider scope than just the code samples relationship to normal Java code is taken however, the purpose of the exercise reveals itself. The internal DSL allows for the abstraction of expressing what the user wants away from normal object creation. Users can now think in terms of "computers" which have "processors" and "disks" and allow for the internal DSL's workings to create the object they need for them. This approach can be applied to the construction of graph representation of protocols, allowing for users to place the structure of the graph at the heart of their reasoning, separate from object construction.

An example of the API DynoSesh exposes for constructing protocols is shown in listing 7. This will result in a protocol of two nodes, with one start node that accepts a single `TestClass` sent over the protocol from actor 0, and then another node which will recursively accept more `TestClass` objects from actor 0 infinitely. During the construction of a protocol, nodes that are declared earlier in the process need to be able to potentially access those which are defined later, therefore no connections between objects are instantiated until the `NodeBuilder.build()` method is called to create the protocol. There are a number of error checking procedures controlling protocol construction to ensure that invalid protocols cannot be produced, these include but are not limited to ensuring that a single protocol cannot have more than one start node.

Listing 7: Construction of a DynoSesh graph using the internal DSL

```
1  ProtocolBuilder protocolBuilder = new ProtocolBuilder();
2  Protocol protocol = protocolBuilder
3          .node()
4                  .payload(null)
5                  .connection()
6                          .actor(``0'')
7                          .to(``1'')
8          .node()
9          .payload(TestClass.class)
10                 .connection()
11                         .actor(``0'')
12                         .to(``1'')
13         .build();
```

The internal DSL also allows for the creation of branches, with one node able to have multiple connections, there are a number of paths that communication can take throughout the protocol graph depending on what transmissions occur.

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] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty asynchronous session types". In: *ACM SIGPLAN Notices* 43.1 (2008), pp. 273–284.

[8] 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.

[9] Raymond Hu, Nobuko Yoshida, and Kohei Honda. "Session-based distributed programming in Java". In: *European Conference on Object-Oriented Programming*. Springer. 2008, pp. 516–541.

[10] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. "Session types for Rust". In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. ACM. 2015, pp. 13–22.

[11] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. "SPY: local verification of global protocols". In: *International Conference on Runtime Verification*. Springer. 2013, pp. 358–363.

[12] 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.

[13] Akka Development Team. *Akka*. 2018. URL: http://akka.io (visited on 06/13/2018).

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

[15] B Clifford Neuman and Theodore Ts'o. "Kerberos: An authentication service for computer networks". In: *IEEE Communications magazine* 32.9 (1994), pp. 33–38.

[16] Taher Elgamal and Kipp EB Hickman. *Secure socket layer application program apparatus and method*. US Patent 5,657,390. 1997.

[17] Mike Bland. "Finding more than one worm in the apple". In: *Queue* 12.5 (2014), p. 10.

[18] Martin Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321712943, 9780321712943.