Session-Based Typechecking for Elixir Modules Using ElixirST

Adrian Francalanza Gerard Tabone
Dept. of Computer Science
University of Malta
Msida, Malta
{adrian.francalanza, gerard.tabone}@um.edu.mt

We investigate the adaptation of session types to provide static behavioural guarantees for Elixir modules. We devise a type system, called ElixirST, which allows us to describe the behaviour of executing a public function in an Elixir module as a protocol of message interactions. The ElixirST type system also allows us to statically determine whether this function observes its endpoint specification; a corresponding tool automating the corresponding typechecking on Elixir source code has also been constructed. We also formally validate this type system. An LTS-based operational semantics for the language fragment supported by the type system is developed, modelling its runtime behaviour when interacting with an arbitrary module client that respects a compatible protocol. This operational semantics is then used to prove a form of session fidelity and progress for ElixirST. The proposed talk will give an overview of the tool and its underlying theory, which was recently published in [4].

Introduction Elixir [8, 9] is a functional programming language built on the Erlang ecosystem. It is renowned for providing a robust foundation for building concurrent programs via the actor model. Elixir programs are structured as a collection of *modules* that contain *functions*, the basic unit of code decomposition in the language. A module only exposes a subset of these functions to external invocations by defining them as *public*; these functions act as the only entry points to the functionality encapsulated by the module where they are defined. Internally, the bodies of these public functions may then invoke other functions, which can either be the *public* ones already exposed or else *private* functions that can only be invoked from within the same module.

A prevalent design pattern employed for Elixir modules is the *service handler*. In this pattern, a server process (actor) listens for client requests received as messages containing the process ID of client (another process) as part of the payload. For each request, the server spawns a new process executing a (public) function from the module to act as a dedicated client handler that executes separately from the server. After the respective process IDs of the client and the spawned handler are made known to each other, a session of interaction commences between the two concurrent entities (via message-passing).

The service handler pattern brings about benefits that improve software quality, *e.g.* it induces scalability (*i.e.,* the server is free to handle further requests from other clients) and fault locality (*i.e.,* if the interaction between the handler and the client goes wrong, it is limited to that session of interaction). However, by itself (and the existing Elixir support), it does not provide mechanisms to statically rule out a variety of errors. More concretely, traditional interface elements such as function parameters (used to instantiate the executing function body with values such as the client process ID) and the function return value (reporting the eventual outcome of handled request) only assist with correct function calls, but do not cover the correctness of the messages exchanged between the two concurrent parties within a session. Communication incompatibilities between the interacting parties could lead to various runtime errors. For instance, if in a session a message is sent with an unexpected payload, it could cause the receiver's subsequent computation depending on it to crash (*e.g.* multiplying by a string when a number

Submitted to:
30 Years of Session Types

should have been received instead). Also, if messages are exchanged in an incorrect order, they may cause *deadlocks* (*e.g.* two processes waiting forever for one another to send messages of a particular kind when a message of a different kind has been sent instead).

As a potential solution to enhance static guarantees for situations such as the service handler, we propose the adaptation of session types in order to regulate the communication protocol between the handler and the client. The adaptation should however satisfy several criteria. First, the burden of the session type annotation in Elixir code should be minimised. Second, session-typing should not interfere with existing Elixir idioms and code structuring principles but complement existing static checks. Third, typechecking should operate in a setting where only part of the codebase is available, namely that of the server module; the associated static guarantees should be also tailored to such a setting. These factors sets our proposal apart from existing solutions for actor-based systems.

Contribution We present a type-checker to assist with communication side-effects within an Elixir module in two ways: (a) it allows module designers to formalise the session endpoint protocol as a session type, and ascribe it *exclusively* to public functions; (b) it statically verifies whether the body of a function respects the ascribed session type protocol specification. The type-checker does not require private functions to be annotated, even though they can be invoked by the annotated public counterparts. Moreover, it typechecks the module without requiring access to the client code invoking the module public functions. Although this provides greater flexibility since it is often unrealistic to have both client and handler code-bases available for static analysis, this complicates the formalisation of the guarantees provided by session type checking. To this end, we formalise the runtime semantics of the Elixir language fragment supported by ElixirST as a labelled transition system (LTS), modelling the execution of a spawned handler interacting with a client within a session (left implicit). This operational semantics then allows us to prove a conditional form of the *session fidelity* and *progress* properties for ElixirST.

Tool The code for the type-checker, called ElixirST, is available at:

https://github.com/gertab/ElixirST

The key part of our tool is that it does not interfere with existing idioms within a language, but rather reuses existing mechanisms provided by the Elixir language, *e.g.* `@spec` annotations. Concretely, our implementation integrates seamlessly within this compilation pipeline, where the developer decorates functions with a specific session type using annotations, such as the following.

```
@session "X = !ping(number).?pong(number).X"
@spec f(number) :: atom
def f(x) do ... end
```

During compilation, these annotations are used to enforce that a function abides by the given protocol, ensuring that a process communicating with such a process will not run into errors during runtime.

Related Work Statically typing the structure of actors is hard due to the dynamic nature the processes and their incoming messages. Several implementations [2, 6, 7] dynamically monitor actors interactions. Other attempts analyse actors statically via either a custom language design [5] or by typing mailboxes directly [1, 3]. ElixirST differs as it aims to retrofit session types into an existing language, supporting its existing design patterns.

Conclusion In this demo/talk we plan to give a brief overview of the design of the type system, along with its formal properties, recently presented in [4], and demonstrate how the tool works using some example Elixir programs.

References

[1] Ugo de'Liguoro & Luca Padovani (2018): *Mailbox Types for Unordered Interactions*. In Todd D. Millstein, editor: *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, *LIPIcs* 109, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 15:1–15:28, doi:10.4230/LIPIcs.ECOOP.2018.15.

[2] Simon Fowler (2016): *An Erlang Implementation of Multiparty Session Actors*. In Massimo Bartoletti, Ludovic Henrio, Sophia Knight & Hugo Torres Vieira, editors: *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016*, *EPTCS* 223, pp. 36–50, doi:10.4204/EPTCS.223.3.

[3] Simon Fowler, Duncan Paul Attard, Franciszek Sowul, Simon J Gay & Phil Trinder (2023): *Special Delivery: Programming with Mailbox Types*. *Proc. ACM Program. Lang.* (ICFP).

[4] Adrian Francalanza & Gerard Tabone (2023): *ElixirST: A Session-Based Type System for Elixir Modules*. *Journal of Logical and Algebraic Methods in Programming* 135, p. 100891, doi:10.1016/j.jlamp.2023.100891.

[5] Paul Harvey, Simon Fowler, Ornela Dardha & Simon J. Gay (2021): *Multiparty Session Types for Safe Runtime Adaptation in an Actor Language*. In Anders Møller & Manu Sridharan, editors: *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, *LIPIcs* 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 10:1–10:30, doi:10.4230/LIPIcs.ECOOP.2021.10.

[6] Rumyana Neykova & Nobuko Yoshida (2017): *Let It Recover: Multiparty Protocol-Induced Recovery*. In: *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, Association for Computing Machinery, New York, NY, USA, p. 98–108, doi:10.1145/3033019.3033031.

[7] Rumyana Neykova & Nobuko Yoshida (2017): *Multiparty Session Actors*. *Log. Methods Comput. Sci.* 13(1), doi:10.23638/LMCS-13(1:17)2017.

[8] The Elixir Team (2023): *Elixir Documentation*. Available at `https://elixir-lang.org/docs.html`.

[9] Dave Thomas (2018): *Programming Elixir: Functional, Concurrent, Pragmatic, Fun*. Pragmatic Bookshelf.