Session Types in Elixir

Gerard Tabone
gerard.tabone.17@um.edu.mt
University of Malta
Department of Computer Science
Msida, Malta

Adrian Francalanza
adrian.francalanza@um.edu.mt
University of Malta
Department of Computer Science
Msida, Malta

Abstract

This paper proposes an adaptation of session types to provide behavioural information about public functions in Elixir modules. We formalise typechecking rules for the main constructs of the language. This allows us to statically determine whether a function implementation observes its session endpoint specification. Based on this type system, we then construct a tool that automates typechecking for Elixir modules.

CCS Concepts: • **Theory of computation** → *Concurrency*; *Program analysis*; • **Software and its engineering** → Software verification and validation.

Keywords: session types, concurrency, Elixir, functional programming

ACM Reference Format:
Gerard Tabone and Adrian Francalanza. 2021. Session Types in Elixir. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE '21), October 17, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3486601.3486708

1 Introduction

The Elixir [41] programming language is a dynamically typed, actor-based, functional language built on top of the Erlang ecosystem [3]. It provides a modern Ruby-like syntax while preserving most of Erlang's concurrency features, taking advantage of the BEAM's well-established foundation. Elixir concurrency permits computation to be split across multiple actors [1] called processes. In Elixir, processes are *spawned* by other processes to execute independently using a local memory. Every process, when spawned, is dynamically assigned a unique process identifier (*pid*), which is used by other processes to interact with it by sending it *asynchronous messages*. These messages are received at an incoming message buffer called the *mailbox*, from where they can be selectively read by the addressee process. Elixir processes are very lightweight, and it is commonplace to have thousands of them running concurrently on the same machine.

Even though the actor model disciplines concurrency, Elixir programs are still susceptible to subtle concurrency bugs that are hard to detect and reproduce. The language provides support for detecting errors relating to the *functional part* of the language: calling a function with incorrect type parameters can be detected at compile-time via **`@spec`** annotations and tools such as the Dialyzer [25]. However, the execution of Elixir functions is often side-effectful due to messaging, and existing tools are limited in this regard. For example, differences in the types of messages that can be received by a process and in their order of arrival can clearly influence the behaviour of a process.

This paper proposes the use of *session types* [2, 17, 43] as a means for statically checking and verifying communicating Elixir programs. In its simplest form, a (binary) session type defines a protocol between two processes, consisting of *send* and *receive* statements, coupled with choices, recursion and termination. Session types are used to guarantee that each *send* statement matches a corresponding *receive* statement (and vice versa) and can lead to systems that are free from behavioural errors such as deadlocks and protocol infidelity. Session types have been widely adapted in several programming languages, either as static checkers (*e.g.*, Rust [22, 24], Scala [38], Go [8] and OCaml [20, 34]) or as runtime monitors (*e.g.*, Python [32], Erlang [11], OCaml [28] and Scala [5]). In this work, we apply session types to Elixir as a static checker.

Contribution. We propose a method to augment Elixir modules with static information about the message interactions of public functions within that module. Our proposed method reuses existing mechanisms offered by Elixir, which facilitates its integration within existing language development workflows. The augmented static information is expressed in terms of a (binary) session type, using the designated **`@session`** annotation; together with the **`@spec`** annotations, session type specifications act as a behavioural API for the functions proffered by the module. We demonstrate the realisability of our proposal by building a static type checker that can verify whether an annotated public

```
1   defmodule Counter do
2
3   @spec server(pid, number) :: atom
4   def server(client, tot) do
5     receive do
6        {:incr, val} -> server(client, tot + val)
7        {:stop}      -> terminate(client, tot)
8     end
9   end
10
11  @spec terminate(pid, number) :: atom
12  defp terminate(client,tot) do
13    send(client, {:value, tot})
14    :ok
15  end
16
17  end # module
```

Listing 1. Counter written in Elixir

function adheres to the communication protocol specified by the **@session** and **@spec** annotations.

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

https://github.com/gertab/ElixirST

Roadmap. Sec. 2 provides an example in Elixir and motivates the use of session types. Sec. 3 describes the formal view, including the typing rules of our type checker. Sec. 4 explores the design details of our tool. Finally, we discuss the related work in Sec. 5.

2 Motivating Example

Consider a simple *counter* system, adapted from [33], whereby a (server) process stores a counter total which can be increased by a (client) interacting process or else terminated by this same (client) process. A sample Elixir `Counter` module is shown in Listing 1. It offers one public function called `server` on lines 4–9 taking two arguments: the *pid* of the client, `client`, and the initial counter total, `tot`. A process executing this function waits to receive client requests as messages in its mailbox using the **`receive do ... end`** statement; this construct is blocking, meaning that the process stops until a message with the expected format is received. The `server` function accepts two types of messages, namely, increment requests with label `:incr` carrying payload `val`, or termination requests denoted by the label `:stop`. This function branches accordingly: for increment requests, it recurses while updating the running total to `tot+val` on line 6, whereas termination requests on line 7 are handled by calling the *private* function `terminate`. Private functions, defined using **`defp`**, are only visible from *within* a module. In this case, the function `terminate` (defined on lines 12–15) sends a `:value` message carrying the final total value `tot` to the `client` process and terminates with the atom value `:ok`. Assuming that a client process carrying a *pid* bound to

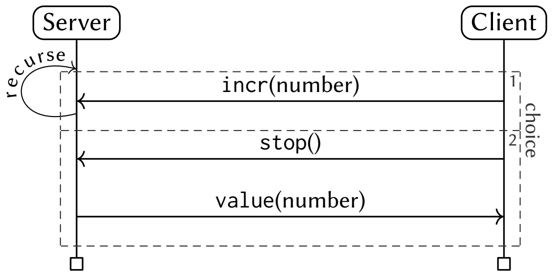

Figure 1. Counter protocol

variable `cid` already exists, a counter server linked to `cid` initialised with a running total of `0` can be launched using the statement:

```
sid = spawn(Counter, :server, [cid, 0]).
```

Elixir conducts dynamic typechecking to catch runtime errors. In addition, **`@spec`** annotations such as those on lines 3 and 11 can help with detecting potential errors at compile-time. However, the language offers limited support to assist the static detection of errors relating to the concurrent messaging. For instance, it might not be immediately apparent that the payload carried by a `:incr` request should be a `number` value. Similarly, the code in Listing 1 does not necessarily convey the information that the intended interaction with a `server` process should follow the protocol depicted in Fig. 1. This abstract specification states that a server can be incremented an arbitrary number of times, followed by a single termination request (*i.e.,* no further increment or termination requests can succeed it).

From the perspective of the server, the entire session of interactions can be formalised as the *session type* (called *counter*) below:

$$counter = \&\Big\{?\mathtt{incr}(\mathsf{number}).counter,\quad ?\mathtt{stop}().!\mathtt{value}(\mathsf{number}).\mathsf{end}\Big\} \tag{1}$$

The type states that the server can *branch* (*i.e.,* &) in two ways: if it receives (*i.e.,* ?) an `incr` label with a number payload, the server recurses back to the beginning; and if it receives a `stop` label, it has to send (*i.e.,* !) back a label `value` with a payload of type number (*i.e.,* !value(number)). No further interactions are allowed when the end statement is reached. Accordingly, the client has to follow the *dual* of the same session type.

$$\overline{counter} = \oplus\Big\{!\mathtt{incr}(\mathsf{number}).\overline{counter},\quad !\mathtt{stop}().?\mathtt{value}(\mathsf{number}).\mathsf{end}\Big\} \tag{2}$$

Concretely, it can repeatedly make a *choice* (*i.e.,* ⊕) to send one of two labels, either `increment` or `stop`. The former ensures that it recurses back to the beginning, while the latter results in the client receiving a value of type number.

This paper proposes an approach whereby module definitions are augmented with a **`@session`** annotation for *public*

```
1   defmodule Counter do
2
3   @session "counter = &{?incr(number).counter,
4                        ?stop().!value(number).end}"
5   @spec server(pid, number) :: atom
6   def server(client, value) do
7     ...
8   end
9
10  @spec terminate(pid, number) :: atom
11  defp terminate(client,tot) do
12    ...
13  end
14
15  ...
16
17  @dual "counter"
18  @spec client(pid) :: number
19  def client(server) do
20     ...
21  end
22
23  end # module
```

Listing 2. Counter annotated with *session types*

functions, as shown in Listing 2. Whereas line 3 requires the function `server` to adhere to the session type *counter*, no session annotation is required for the private function `terminate` on line 11. Lines 19–21 present a case in which the client code is defined as a public function within the same module; in such a case, we can annotate it with the **`@dual`** information on line 17.

Our proposed session type annotations serve two important purposes. On the one hand, they provide a high-level (yet formal) specification as to how a public function is to be interacted with, without the need to look inside its implementation, as in the case of line 3. *E.g.*, by glancing at the *counter* session type on line 3 of Listing 2, one can immediately tell that a process running function `server` accepts two types of messages with labels `incr` or `stop`. On the other hand, they allow function implementations to be typechecked against such specifications. *E.g.*, we are able to statically ascertain that the function `server` (and its ancillary function `terminate`) adheres to the protocol dictated by session type `counter` on line 3. We can also reject the problematic `client` implementation given in Listing 3 at compile-time, on the grounds that it violates the dual session type of *counter*. Concretely, the client selects an illegal choice `decr` on line 21; it also expects to receive a value with a number (line 24) after 'forgetting' to send a termination request (*i.e.,* a message with a `stop` label). Both cases breach the $\overline{\textit{counter}}$ protocol.

3 A Formal Analysis

In this section we provide a formalisation of our approach that underpins the typechecking tool presented in Sec. 4.

```
19  def client(server) do
20     send(server, {:incr, 5})
21     send(server, {:decr, 2})
22     # send(server, {:stop})
23
24     receive do
25        {:value, num} -> num
26     end
27  end
```

Listing 3. Counter client with issues

We outline the Elixir syntactic subset supported by our tool (Sec. 3.1), together with the behavioural type system used by our static type checker (Sec. 3.2).

3.1 Elixir Syntax

Fig. 2 presents a fragment of the Elixir syntax, including its communication primitives and the new annotations. We let x range over variable names, m over module names, f over function names and $\mathtt{l}$ over labels.

Session Types. As previewed in Sec. 2 our analysis will rely on an interpretation of session types, adapted to (a subset of) Elixir. The syntax is defined by the grammar in Fig. 2 and assumes a standard set of *expression types*, T, also defined in Fig. 2. Expression types consist of base types boolean, number, atom and pid, together with the inductive types for tuples and lists.

Session types consist of the branching, choice, recursion and termination constructs. The branching session type $\&\left\{?\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\right\}_{i\in I}$ describes an interaction that receives a message containing any one of the labels $\mathtt{l}_i$ and respective payloads of type $\widetilde{T_i}$, and proceeds as S_i. The choice type $\oplus\left\{!\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\right\}_{i\in I}$ describes an interaction that sends a message indexed by any one of the labels $\mathtt{l}_i$ with a payload of type $\widetilde{T_i}$ and continues as S_i. In both branches and choices, labels $\mathtt{l}_i$ are assumed to be unique. In the case of singleton choices or branching types, the $\oplus$ and $\&$ can be omitted, *e.g.*, $?\mathtt{hello}().S$ can be used as the shorthand for the session type $\&\left\{?\mathtt{hello}().S\right\}$. The recursion type $\mathsf{rec}\ \mathsf{X}\,.\,S$ binds recursion variable X in continuation S. We adopt an *equi-recursive* [35] approach, so $\mathsf{rec}\ \mathsf{X}\,.\,S$ and its unfolding $S\left[{}^{\mathsf{rec}\,\mathsf{X}\,.\,S}/_{\mathsf{X}}\right]$ can be used interchangeably. A session terminates when an end is reached, which can also be omitted for brevity. The *dual* type of S, denoted as $\overline{S}$, is defined as follows:

$$\overline{\&\left\{?\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\right\}_{i\in I}} = \oplus\left\{!\mathtt{l}_i\big(\widetilde{T_i}\big).\overline{S_i}\right\}_{i\in I} \qquad \overline{\mathsf{X}} = \mathsf{X} \quad \overline{\mathsf{end}} = \mathsf{end}$$

$$\overline{\oplus\left\{!\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\right\}_{i\in I}} = \&\left\{?\mathtt{l}_i\big(\widetilde{T_i}\big).\overline{S_i}\right\}_{i\in I} \qquad \overline{\mathsf{rec}\ \mathsf{X}\,.\,S} = \mathsf{rec}\ \mathsf{X}\,.\,\overline{S}$$

Modules and Functions. An Elixir program is defined as a module of the form `defmodule` m `do` $\widetilde{P}\ \widetilde{D}$ `end`. A module takes a sequence of private ($\widetilde{P}$) and public ($\widetilde{D}$) functions; by convention, $\widetilde{P}$ stands for the possibly empty sequence $P_1 \ldots P_n$ (similar for $\widetilde{D}$). Functions are named, f, and their

$$
\begin{array}{rrcl}
\text{Session types} & S & ::= & \&\big\{?\mathtt{l}_i(\widetilde{T_i}).S_i\big\}_{i\in I} \mid \oplus\big\{!\mathtt{l}_i(\widetilde{T_i}).S_i\big\}_{i\in I} \\
 & & \mid & \mathsf{rec}\ \mathsf{X}\,.\,S \mid \mathsf{X} \mid \mathsf{end} \\
\text{Expression types} & T & ::= & \mathsf{boolean} \mid \mathsf{number} \mid \mathsf{atom} \mid \mathsf{pid} \\
 & & \mid & \{T_1,\ \ldots,\ T_n\} \mid [\,T\,] \\
 & & & \\
\text{Module} & M & ::= & \mathtt{defmodule}\ m\ \mathsf{do}\ \widetilde{P}\ \widetilde{D}\ \mathsf{end} \\
\text{Function} & D & ::= & K \quad B \quad \mathtt{def}\ f(\widetilde{x})\ \mathsf{do}\ t\ \mathsf{end} \\
\text{Private function} & P & ::= & B \quad \mathtt{defp}\ f(\widetilde{x})\ \mathsf{do}\ t\ \mathsf{end} \\
\text{Type ann.} & B & ::= & \mathtt{@spec}\ f(\widetilde{T})\ ::\ T \\
\text{Session ann.} & K & ::= & \mathtt{@session}\ \text{“}\mathsf{X} = S\text{”} \mid \mathtt{@dual}\ \text{“}\mathsf{X}\text{”} \\
 & & & \\
\text{Basic values} & b & ::= & \mathit{boolean} \mid \mathit{number} \mid \mathit{atom} \mid \mathit{pid} \mid [\ ] \\
\text{Values} & v & ::= & b \mid [\,v_1 \mid v_2\,] \mid \{v_1,\ \ldots,\ v_n\} \\
\text{Identifiers} & w & ::= & b \mid x \\
\text{Patterns} & p & ::= & w \mid [\,w_1 \mid w_2\,] \mid \{w_1,\ \ldots,\ w_n\} \\
\text{Terms} & t & ::= & e \\
 & & \mid & x = t_1;\ t_2 \\
 & & \mid & \mathsf{send}\,(x, \{:\mathtt{l}, e_1,\ \ldots,\ e_n\}) \\
 & & \mid & \mathtt{receive\ do} \\
 & & & \quad \big(\{:\mathtt{l}_i, p_i^1,\ \ldots,\ p_i^n\} \to t_i\big)_{i\in I}\mathsf{end} \\
 & & \mid & f\,(e_1,\ \ldots,\ e_n) \\
 & & \mid & \mathsf{case}\ e\ \mathsf{do}\ (p_i \to t_i)_{i\in I}\mathsf{end} \\
 & & & \\
\text{Expressions} & e & ::= & w \mid e_1 \diamond e_2 \mid \mathsf{not}\ e \\
 & & \mid & e_1\ \mathsf{and}\ e_2 \mid e_1\ \mathsf{or}\ e_2 \\
 & & \mid & [\,e_1 \mid e_2\,] \mid \{e_1,\ \ldots,\ e_n\} \\
\text{Operators} & \diamond & ::= & < \mid > \mid <= \mid >= \mid == \mid != \\
 & & \mid & + \mid - \mid * \mid /
\end{array}
$$

Figure 2. Elixir syntax

body, t, is parameterised by a variable sequence $\tilde{x}$ that defines its arity. For instance, in the case of Listing 2, the public function `server` and the private function `terminate` have arity 2, whereas the public function `client` has arity 1; Elixir modules can have multiple functions with the same name, as long as they have distinct arities. For every public function $\mathsf{def}\ f(\widetilde{x})\ \mathsf{do}\ t\ \mathsf{end}$ and private function $\mathtt{defp}\ f(\widetilde{x})\ \mathsf{do}\ t\ \mathsf{end}$, every variable x_i in the sequence of parameters $\widetilde{x} = x_1,\ \ldots,\ x_n$ must be unique. Functions are annotated by their specification type (`@spec`), denoted as B in Fig. 2. It takes the form $f(\widetilde{T}) :: T$, where the parameters are assigned types $\widetilde{T}$ and the return type is T. In addition, public functions are also annotated with session types, K in Fig. 2. These can be either stated directly, `@session` “X = S”, or indirectly, `@dual` “X”, whenever the session name X is defined within the same module. The definition $\mathsf{X} = S$ is equivalent to the session type $\mathsf{rec}\ \mathsf{X}\,.\,S$, but it also allows the session type $\mathsf{rec}\ \mathsf{X}\,.\,S$ to be referenced from within the module as the name X.

Terms and Expressions. A term t can take the form of either an expression e, a *let* statement, a send statement, a receive construct, a case statement or a function call. The *let* statement $x = t_1;\ t_2$ evaluates t_1 and binds its result to x in t_2. We can write $t_1;\ t_2$ as syntactic sugar for $x = t_1;\ t_2$ where x is a *fresh* variable. The term $\mathsf{send}\,(x, \{:\mathtt{l}, e_1,\ \ldots,\ e_n\})$ sends a message to x (denoting the *pid* of some process); the message consists of a tuple where the *literal* atom in the first position of the tuple acts as a label for the message. The term $\mathtt{receive\ do}\ \big(\{:\mathtt{l}_i, p_i^1,\ \ldots,\ p_i^n\} \to t_i\big)_{i\in I}\mathsf{end}$ allows a process to receive a message and proceed as term t_i. The message has to match using the label and the patterns p via pattern matching. A pattern p can take the form of a variable, basic value (*e.g.*, *true*), tuple (*e.g.*, $\{x, y\}$) or list (*e.g.*, $[\,x \mid y\,]$ where x is the head and y is the tail). All variables within a pattern are assumed to be unique. The term $\mathtt{case}\ e\ \mathtt{do}\ (p_i \to t_i)_{i\in I}\mathsf{end}$ matches e with the patterns p_i and proceeds as t_i. A function call takes the form of $f\,(e_1,\ \ldots,\ e_n)$.

An expression e can be a variable, basic values (*i.e., booleans*, *numbers*, *atoms*[1] and *pids*) or other operations. The latter include arithmetic (+, −, ∗, /), comparison (<, >, <=, >=, ==, !=) and boolean operations (*and*, *or*, *not*). An expression can also take the form of a list of expressions (all elements having the same type) or a tuple (where elements can have different types).

3.2 Session Typing

The module and term typing rules are defined in Figs. 3 and 4, and use three types of environments:

$$
\begin{aligned}
\Gamma &::= \emptyset \mid \Gamma, x : T \\
\Delta &::= \emptyset \mid \Delta, f_n : S \\
\Sigma &::= \emptyset \mid \Sigma, f_n : \left\{\begin{matrix} \mathtt{params} = \widetilde{x},\ \mathtt{param\_types} = \widetilde{T}, \\ \mathtt{body} = t,\ \mathtt{return\_type} = T \end{matrix}\right\}
\end{aligned}
$$

The *variable binding* environment Γ maps variables to expression types ($x : T$). The *function type* environment Δ maps function names (and arity) to their session type ($f_n : S$) and is used to typecheck recursive calls. The *function information* environment Σ is a static environment which stores information regarding the functions in a module. When a function named f with arity n is defined in a module, $\Sigma(f_n)$ returns environment which stores the parameter names $\widetilde{x}$ and their types $\widetilde{T}$, the function body t and the return type T.

[1] An atom is defined as a colon followed by a label (:l), *e.g.*, :dog.

$$\boxed{\vdash M}$$

$$\textsc{tModule}\ \dfrac{\begin{array}{c}\forall f_n \in \mathtt{functions}(\widetilde{D}) \quad \Sigma_{f_n}(f_n) = \sigma \quad \sigma.\mathtt{params} = \widetilde{x} \quad \sigma.\mathtt{param\_types} = \widetilde{T} \quad \sigma.\mathtt{body} = t \quad \sigma.\mathtt{return\_type} = T \\ \Sigma = \mathtt{details}(\widetilde{P}) \qquad S = \mathtt{session}\,(f_n) \qquad (f_n : S) \cdot (\widetilde{x} : \widetilde{T}) \vdash_{\Sigma \cup \Sigma_{f_n}} S \rhd t : T \lhd \mathtt{end}\end{array}}{\vdash \mathtt{defmodule}\ m\ \mathtt{do}\ \widetilde{P}\ \widetilde{D}\ \mathtt{end}}$$

Figure 3. Module typing

$$\boxed{\Delta \cdot \Gamma \vdash_\Sigma S \rhd t : T \lhd S'}$$

$$\textsc{tLet}\ \dfrac{\Delta \cdot \Gamma \vdash S \rhd t_1 : T \lhd S' \qquad \Delta \cdot (\Gamma, x : T) \vdash S' \rhd t_2 : T' \lhd S''}{\Delta \cdot \Gamma \vdash S \rhd x = t_1;\ t_2 : T' \lhd S''}$$

$$\textsc{tBranch}\ \dfrac{\forall i \in I \qquad \forall j \in 1..n \qquad \vdash_{\mathrm{pat}} p_i^j : T_i^j\ \rhd\ \Gamma_i^j \qquad \Delta \cdot \left(\Gamma, \Gamma_i^1,\ \ldots,\ \Gamma_i^n\right) \vdash S_i \rhd t_i : T \lhd S'}{\Delta \cdot \Gamma \vdash \&\left\{?\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\right\}_{i \in I} \rhd \mathtt{receive\ \ do}\ \left(\left\{:\mathtt{l}_i, \widetilde{p_i}\right\} \longrightarrow t_i\right)_{i \in I \cup J}\mathtt{end} : T \lhd S'}$$

$$\textsc{tChoice}\ \dfrac{\exists i \in I \qquad \mathtt{l} = \mathtt{l}_i \qquad \Gamma \vdash_{\mathrm{exp}} x : \mathtt{pid} \qquad \forall j \in 1..n \qquad \Gamma \vdash_{\mathrm{exp}} e_j : T_i^j}{\Delta \cdot \Gamma \vdash \oplus\left\{!\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\right\}_{i \in I} \rhd \mathtt{send}\,(x, \{:\mathtt{l}, e_1,\ \ldots,\ e_n\}) : \left\{\mathtt{atom}, T_i^1,\ \ldots,\ T_i^n\right\} \lhd S_i}$$

$$\textsc{tRecUnknownCall}\ \dfrac{\begin{array}{c}\Sigma\,(f_n) = \sigma \quad \sigma.\mathtt{params} = \widetilde{x} \quad \sigma.\mathtt{param\_type} = \widetilde{T} \quad \sigma.\mathtt{body} = t \quad \sigma.\mathtt{return\_type} = T \\ f_n \notin \mathit{dom}(\Delta) \qquad (\Delta, f_n : S) \cdot \left(\Gamma, \widetilde{x} : \widetilde{T}\right) \vdash S \rhd t : T \lhd S' \qquad \forall i \in 1..n \quad \Gamma \vdash_{\mathrm{exp}} e_i : T_i\end{array}}{\Delta \cdot \Gamma \vdash S \rhd f\,(e_1,\ \ldots,\ e_n) : T \lhd S'}$$

$$\textsc{tRecKnownCall}\ \dfrac{\begin{array}{c}\Sigma\,(f_n) = \sigma \qquad \sigma.\mathtt{return\_type} = T \qquad \sigma.\mathtt{param\_types} = \widetilde{T} \\ \Delta\,(f_n) = S \qquad \forall i \in 1..n \qquad \Gamma \vdash_{\mathrm{exp}} e_i : T_i\end{array}}{\Delta \cdot \Gamma \vdash S \rhd f\,(e_1,\ \ldots,\ e_n) : T \lhd \mathtt{end}}$$

$$\textsc{tCase}\ \dfrac{\begin{array}{c}\Gamma \vdash_{\mathrm{exp}} e : U \\ \forall i \in I \qquad \vdash_{\mathrm{pat}} p_i : U\ \rhd\ \Gamma_i' \qquad \Delta \cdot \left(\Gamma, \Gamma_i'\right) \vdash S \rhd t_i : T \lhd S'\end{array}}{\Delta \cdot \Gamma \vdash S \rhd \mathtt{case}\ e\ \mathtt{do}\ (p_i \longrightarrow t_i)_{i \in I}\mathtt{end} : T \lhd S'} \qquad \textsc{tExpression}\ \dfrac{\Gamma \vdash_{\mathrm{exp}} e : T}{\Delta \cdot \Gamma \vdash S \rhd e : T \lhd S}$$

Figure 4. Term typing

A program is typechecked by starting from a module and then checking that each public function behaves according to the pre-defined session type, as shown in tModule. This rule uses some auxiliary functions: $\mathtt{details}(\widetilde{P})$ populates the environment Σ; $\mathtt{functions}(\widetilde{D})$ returns a set of all public function names; and $\mathtt{session}(f_n)$ obtains the session type for f_n (*i.e.,* the function f with arity n), either directly from `@session` or by computing the dual session type whenever `@dual` is used. tModule uses Σ_{f_n}, which contains a mapping of the public function f_n to its function information, used again when typechecking recursion. Each public function is then checked using the term typing rules in Fig. 4. These rules are syntax directed, having at most one rule for each term t. The term typing judgement has the form

$$\Delta \cdot \Gamma \vdash_\Sigma S \rhd t : T \lhd S'$$

which can be read as, "the term t can produce a value of type T after following an interaction protocol starting from the initial session type S up to the residual session type S', subject to the *session typing* environment Δ, *variable binding* environment Γ and *function information* environment Σ". Since the function information environment is static throughout a term typing derivation, it is left implicit in the rules of Fig. 4.

The most complex term typing rule in Fig. 4 is tBranch. A `receive` statement expects a branching session type $\&\{\ldots\}$. Each branch from the session type has to match with a `receive` branch, meaning that both the labels and the message types have to match. The message types are obtained using the pattern typing rules in Fig. 6. All branches need to end up with a common expression type T and session type S' — this makes it possible to use the *fork-join* pattern described in Sec. 4.2.

$$\boxed{\Gamma \vdash_{\text{exp}} e : T}$$

$$\textsc{tLiteral}\ \frac{\mathsf{type}\,(b) \ = \ T \qquad b \neq [\,]}{\Gamma \vdash_{\text{exp}} b : T}$$

$$\textsc{tVariable}\ \frac{\Gamma\,(x) = T}{\Gamma \vdash_{\text{exp}} x : T}$$

$$\textsc{tTuple}\ \frac{\forall i \in 1..n \qquad \Gamma \vdash_{\text{exp}} e_i : T_i}{\Gamma \vdash_{\text{exp}} \{e_1,\ \ldots,\ e_n\} : \{T_1,,\ \ldots,\ T_n\}}$$

$$\textsc{tEList}\ \frac{}{\Gamma \vdash_{\text{exp}} [\,]\ :\ [\,T\,]}$$

$$\textsc{tList}\ \frac{\Gamma \vdash_{\text{exp}} e_1 : T \qquad \Gamma \vdash_{\text{exp}} e_2 : [\,T\,]}{\Gamma \vdash_{\text{exp}} [\,e_1 \mid e_2\,] : [\,T\,]}$$

$$\textsc{tArithmetic}\ \frac{\diamond \in \{+,\ -,\ *,\ /\} \qquad \Gamma \vdash_{\text{exp}} e_1 : \mathsf{number} \qquad \Gamma \vdash_{\text{exp}} e_2 : \mathsf{number}}{\Gamma \vdash_{\text{exp}} e_1 \ \diamond\ e_2 : \mathsf{number}}$$

$$\textsc{tComparisons}\ \frac{\diamond \in \{<,\ >,\ <=,\ >=,\ ==,\ !=\} \qquad \Gamma \vdash_{\text{exp}} e_1 : T \qquad \Gamma \vdash_{\text{exp}} e_2 : T}{\Gamma \vdash_{\text{exp}} e_1 \ \diamond\ e_2 : \mathsf{boolean}}$$

$$\textsc{tBoolean}\ \frac{\star \in \{\mathsf{and},\ \mathsf{or}\} \qquad \Gamma \vdash_{\text{exp}} e_1 : \mathsf{boolean} \qquad \Gamma \vdash_{\text{exp}} e_2 : \mathsf{boolean}}{\Gamma \vdash_{\text{exp}} e_1 \ \star\ e_2 : \mathsf{boolean}}$$

$$\textsc{tNot}\ \frac{\Gamma \vdash_{\text{exp}} e : \mathsf{boolean}}{\Gamma \vdash_{\text{exp}} \mathsf{not}\ e : \mathsf{boolean}}$$

Figure 5. Expression typing

$$\boxed{\vdash_{\text{pat}} p : T \ \triangleright\ \Gamma}$$

$$\textsc{tpLiteral}\ \frac{\mathsf{type}\,(b) \ = \ T \qquad b \neq [\,]}{\vdash_{\text{pat}} b : T \ \triangleright\ \emptyset}$$

$$\textsc{tpVariable}\ \frac{}{\vdash_{\text{pat}} x : T \ \triangleright\ x : T}$$

$$\textsc{tpTuple}\ \frac{\forall i \in 1..n \qquad \vdash_{\text{pat}} w_i : T_i \ \triangleright\ \Gamma_i}{\vdash_{\text{pat}} \{w_1,\ \ldots,\ w_n\} : \{T_1,\ \ldots,\ T_n\} \ \triangleright\ \Gamma_1,\ \ldots,\ \Gamma_n}$$

$$\textsc{tpEList}\ \frac{}{\vdash_{\text{pat}} [\,]\ :\ [\,T\,] \ \triangleright\ \emptyset}$$

$$\textsc{tpList}\ \frac{\vdash_{\text{pat}} w_1 : T \ \triangleright\ \Gamma \qquad \vdash_{\text{pat}} w_2 : [\,T\,] \ \triangleright\ \Gamma'}{\vdash_{\text{pat}} [\,w_1 \mid w_2\,] : [\,T\,] \ \triangleright\ \Gamma, \Gamma'}$$

Figure 6. Pattern typing

The rule tChoice expects an initial choice session type $\oplus\{\ldots\}$ which needs to have *exactly* one label matching the message payload of the **send** construct. Similar to tBranch, the respective message data needs to have the correct type, as specified by the session type. Moreover, the **send** operator expects the first parameter to have type pid. Although not explicitly checked, this value represents the addressee of the message and should be equivalent to the *pid* of the dual process in the session.

Our typing prohibits public functions from calling other public functions; they can call any private function as long as the session protocol is respected, and they are also allowed to call themselves to achieve recursive behaviour. Note that these restrictions are introduced in tModule where the only recognised functions are the private functions (in Σ) and the public function itself (in Σ_{f_n}). Private functions are allowed to call any other private function (including themselves) and the originating public function starting the call chain as follows: the first time a function is called, its body needs to be inspected by the type checker. In the case of a public function, this is handled by the premises of the rule tModule in Fig. 3, whereas private function bodies are typed via tRecUnknownCall. Note that, in both cases, the function type environment, Δ, is extended with an entry that maps the function and its arity to the session type. This is important for detecting that the body of the function has already been inspected. More concretely, recursive calls to a function are then handled by the rule tRecKnownCall, which trigger only when the function and arity pair are part of the domain of Δ.

The remaining rules make up the functional aspect of the language. The *let* statement $x = t_1;\ t_2$ is typechecked using the rule tLet. The initial session type S is first transformed to S' due to some actions in t_1 and finally becomes S'' after the actions in t_2. The rule tCase checks the **case** construct, where each case has to match the corresponding type T and session type S. Finally, tExpression checks all expressions e using *expression typing*. Expressions do not have a side effect, so the continuation session type S remains unchanged.

Expression Typing. Expressions are typechecked using the judgement $\Gamma \vdash_{\text{exp}} e : T$, where expression e has type T subject to the variable environment Γ. The expressions typing rules (Fig. 5) are adapted from [7]. Rule tLiteral checks the type of basic values using the function type, which returns the type for any basic value, *e.g.*, $\mathsf{type}\,(\mathit{true}) \ = $ boolean. Rule tVariable checks that variables have the correct type, as specified in Γ. Rules tTuple, tEList and tList check the types of tuples, empty lists and lists, respectively. Rule tArithmetic ensures that numbers are used when doing arithmetic operations. The remaining rules, tBoolean, tNot and tComparisons, are analogous.

Pattern Typing. In the term typing rules tBranch and tCase of Fig. 4, *new* variables are introduced as a result of pattern matching. These need to be assigned to their respective type for typechecking purposes. This is obtained via the judgement $\vdash_{\text{pat}} p : T \triangleright \Gamma$ defined in Fig. 6, which states that all variables in a pattern p are collected (with their type) in Γ. New variables are introduced in tpVariable, and basic values are checked in tpLiteral. Each element in a tuple is checked individually for either values or variables (tpTuple). Lists are checked using tpEList and tpList. Multiple variable environments Γ and Γ' are joined together as Γ, Γ' (their domains must be distinct).

3.3 Typing in Action

Recall the *counter* system from Listing 2, which contains two public functions, `server` and `client`, annotated with the session types, *counter* and $\overline{counter}$, respectively defined in eqs. (1) and (2). We discuss briefly how the type system of Sec. 3.2 can be used to statically analyse this Elixir module.

Typechecking starts from the tModule rule ($\vdash M$, Fig. 3), and the judgement:

$$\vdash \texttt{defmodule Counter do } \widetilde{P}\ \widetilde{D} \texttt{ end}$$

where $\widetilde{D}$ contains the functions `server` and `client`, while $\widetilde{P}$ contains the private function `terminate`. The premise of tModule requires that all public functions are checked individually using the behavioural typing judgement: $\Delta \cdot \Gamma \vdash_{\Sigma} S \triangleright t : T \triangleleft S'$ (Fig. 4). Starting with $f_n = \texttt{server}_2$, the initial session type, S, is set to *counter* and the expected residual session type, S', is end, since functions are only well-typed if they *fully consume* the session type. For the `client` function, the initial session type S is computed to get the dual type of *counter*, given in eq. (2). We focus on the behavioural typing of the `server` function. The function body, t, of `server` consists of the following:

$t =$
```
receive do
  {:incr, val} -> server(client, tot + val)
  {:stop} -> terminate(client, tot)
end
```

This **`receive`** statement is typechecked as the judgement below, using the tBranch rule (from Fig. 4):

$$\Delta \cdot \Gamma \vdash \& \begin{Bmatrix} ?\mathsf{incr}(\mathsf{number}).counter, \\ ?\mathsf{stop}().S_1 \end{Bmatrix} \triangleright t : \mathsf{atom} \triangleleft \mathsf{end}$$

where $S_1 = !\mathsf{value}(\mathsf{number}).\mathsf{end}$.

The session type in tBranch dictates that two branches are required, labelled `incr` and `stop`. The terms inside the branches must match with the continuation session type of the corresponding session type (*i.e., counter* and S_1, respectively). For the first branch (labelled `incr`), the continuation term is a known function call ($\Delta(\texttt{server}_2) = counter$); therefore, we use the tRecKnownCall axiom:

$$\Delta \cdot \Gamma' \vdash counter \triangleright \texttt{server(client, tot+val)} : \mathsf{atom} \triangleleft \mathsf{end}$$

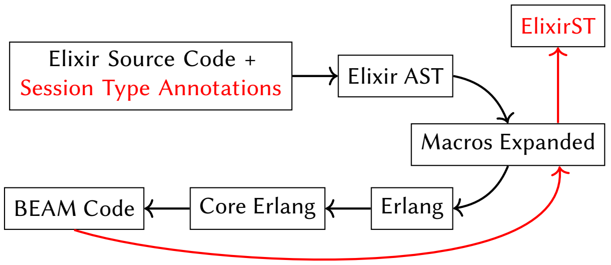

Figure 7. Stages of Elixir compilation along with the session type implementation (in red)

The term of the second branch (labelled `stop`) needs to match with the session type S_1. This branch makes a call to a private function (`terminate`). Since $\texttt{terminate}_1$ is not in the domain of Δ, we proceed to inspect its body using the rule tRecUnknownCall. Recall that private functions are *not* annotated with session types. Accordingly, rule tRecUnknownCall requires us to inherit the outstanding session S_1 as the specification for typing this judgement:

$$\Delta \cdot \Gamma \vdash S_1 \triangleright \texttt{terminate(client, tot)} : \mathsf{atom} \triangleleft \mathsf{end}$$

The derivation of this judgement is left to the interested reader.

As a second example, consider the first two lines of the misbehaving `client` function body from Listing 3, to be typechecked against $\overline{counter}$ from eq. (2):

```
send(server, {:incr, 5})
send(server, {:decr, 2})
```

The first **`send`** statement is checked successfully using the tLet and the tChoice rules. The next **`send`** statement also needs to be checked using tChoice:

$$\Delta' \cdot \Gamma'' \vdash \oplus \begin{Bmatrix} !\mathsf{incr}(\mathsf{number}).\overline{counter}, \\ !\mathsf{stop}().\overline{S_1} \end{Bmatrix} \triangleright \texttt{send(server, \{:decr, 2\})} : \mathsf{number} \triangleleft \mathsf{end}$$

However, the tChoice rule attempts to match `decr` with a nonexistent choice from the session type. Thus, this `client` function is deemed to be ill-typed.

4 Elixir Implementation

This section describes how the type system of Sec. 3 is implemented as the session type checker tool called ElixirST. This tool is integrated in Elixir with minimal changes to the syntax of the surface language.

4.1 Elixir Compilation with Session Types

The Elixir source code is compiled in several steps (see Fig. 7). The original Elixir source code is initially parsed into an Abstract Syntax Tree (AST). Considering that Elixir is a macro-full language, all non-*special form*[2] macros need to be expanded into the *special form* Elixir macros (*e.g.*, **if**/**unless** statements are converted into **case** constructs) [26]. Then, the expanded Elixir AST is converted into Erlang abstract format and Core Erlang. Finally, it is compiled into BEAM code which can be executed on the Erlang Virtual Machine (BEAM).

Our implementation integrates seamlessly within this compilation pipeline (see Fig. 7, red). Inside the Elixir source code, processes are described with a specific session type using annotations (starting with @). Annotations are able to hold information about a module during compile-time. We provide normal labelled session types (**@session**) and their dual (**@dual**, referenced by a label):

```
@session "X = !Ping().?Pong().X"
# ...
@dual "X" # Equivalent to X' = ?Ping().!Pong().X'
```

The annotations set up the *rules* of the session types, which need to be enforced later on in the compilation process. Elixir provides several compile-time hooks which provide a way to alter or append to the compilation pipeline. In this implementation, we initially use the `on_definition` hook to parse the session type (from the annotations) and compute the dual type where required; this is done using the Erlang modules `leex`[3] and `yecc`[4], which create a lexer and a parser, respectively, based on the session type syntax rules shown in Fig. 2. Then, the `after_compile` hook is used to run ElixirST right after the BEAM code is produced. Since the BEAM code stores directly the expanded Elixir AST, ElixirST is able to traverse this AST and verify its concurrent parts using session types.

4.2 Bridging between Elixir and Our Model

Every construct in Fig. 2 maps directly to a corresponding construct in the actual Elixir language. The **@spec** annotation which decorates functions with types is already present in the latest distribution of the language. It is typically used for code documentation and to statically analyse programs using the *Dialyzer* [25], a tool that detects potential (type) errors in Core Erlang programs using success typing [33]. We use the **@spec** information to specify the types for the parameters and the return type of the functions, supplementing our session typechecking analysis. A similar approach to ours was adopted by Cassola et al. [6] for a gradual static type system for Elixir.

[2]*Special form* macros cannot be expanded further, forming the basic building blocks of the Elixir language.
[3]https://erlang.org/doc/man/leex.html
[4]https://erlang.org/doc/man/yecc.html

```
receive do
    {:A} -> send(p, {:C})
            :ok
    {:B} -> send(p, {:C})
            :ok
end
```

```
receive do
    {:A} ->
            :ok
    {:B} ->
            :ok
end
send(p, {:C})
```

The typing rules of Fig. 4 are also designed in a way to minimally alter common coding patterns in the language. For instance, branches in session types might have common continuations, such as $!\mathtt{C}().\mathsf{end}$ in the type $\&\{?\mathtt{A}().!\mathtt{C}().\mathsf{end}, ?\mathtt{B}().!\mathtt{C}().\mathsf{end}\}$. Many type systems force programs to structure their code as shown in the left-hand side code snippet above (which performs the same **send** action twice). However, in Elixir it is common to express this as a fork-join pattern, with a single continuation performing the common action once as shown in the right-hand side code snippet. Our type system can typecheck *both* code snippets.

The only aspect left to discuss is the mechanism used by our implementation to guarantee an interaction between the two processes implementing the respective endpoints of binary session type. To achieve this end, we implemented a bespoke spawning function that takes the code of the respective endpoints and returns a tuple with the *pids* of the two processes that are already linked. The implementation of our session spawn is given below:

```
1   def spawn(leftFn, left_args, rightFn, right_args)
2     when is_function(leftFn)
3                        and is_function(rightFn) do
4     left_pid =
5       spawn(fn ->
6         receive do
7           {:pid, right_pid} ->
8             apply(leftFn, [right_pid | left_args])
9         end
10      end)
11
12    right_pid =
13      spawn(fn ->
14        send(left_pid, {:pid, self()})
15        apply(rightFn, [left_pid | right_args])
16      end)
17
18    {left_pid, right_pid}
19  end
```

This modified `spawn/4` function takes two pairs of arguments: two references of function names (that should be spawned) and their list of arguments. The code implements the initialisation protocol depicted in Fig. 8. It first spawns one process (pre-left in Fig. 8 for line 4) and passes its *pid* to the second spawned process (pre-right in Fig. 8, as the variable `left_pid` on lines 14 and 15). Then, the pre-right process sends its *pid* to the pre-left process (line 14 and lines 6 and 7). At this point, both processes execute their respective functions to transform into the *actual* first and second

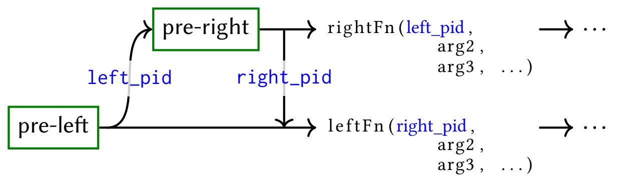
pre-right
pre-left
left_pid
right_pid
rightFn(left_pid, arg2, arg3, ...)
leftFn(right_pid, arg2, arg3, ...)
→ ···

Figure 8. Spawning two processes (green boxes represent *spawned* concurrent processes)

processes participating in the session, passing the respective *pids* as the first argument of the executing functions (lines 8 and 15). In the case of the `Counter` module from Listing 2, we can use the following function call to launch the two processes of the session:

```
ElixirST.spawn(&server/2, [0], &client/1, [])
```

The `spawn/4` function can only launch two processes at a time, in line with the binary sessions. This can however, be easily extended to handle more than two processes in the case of multiparty sessions.

Our implementation still allows spawned processes to receive messages from *any* other process. Unfortunately, unsolicited messages can interfere with a session-typed process, since the receiver is not able to distinguish where the message is originating from in the present implementation. An improvement would be exploiting Elixir's ability to cherry-pick messages out-of-order from the queue using pattern matching. As soon as a session is launched, a unique session ID would be shared with the two parties, and each message exchanged between them would use this ID to identify the source (and destination). This would enable selective reads to filter unsolicited messages. Mostrous and Vasconcelos [29] created a similar system to distinguish messages by attaching a unique reference to each message.

5 Related Work

There are few tools that support development correctness for Elixir. One can, in principle, migrate tools devised for the Erlang ecosystem such as soft-typing checkers [33], model checkers [13] or runtime verifiers [4], but this is rarely done in practice. In fact, most guarantees are usually given via test-driven tools such as ExUnit[5]. To the best of our knowledge, this is the first implementation of session types for the Elixir language. In what follows, we will take a look at session type integration with other languages.

5.1 Session Types on Top of Actor-Based Languages

Python. Neykova and Yoshida [30, 31] introduced (*multiparty*) session types to dynamically typed languages, in this case, Python (flavoured with an actor framework, *Cell*). Using runtime verification, processes are dynamically monitored to ensure that they follow the pre-defined session types written in the Scribble protocol language [18]. Each process is assigned different protocols and roles by the use of Python annotations (*e.g.*, `@role`) which decorate functions similar to our work.

Erlang. Extending the work of Neykova and Yoshida, Fowler [11] created an Erlang library which offers runtime monitoring of actor communication in Erlang, based on multiparty session types. It allows for greater flexibility than our work, since the Erlang/OTP behaviours (*e.g.*, `gen_server`) are used — these coordinate actors in a hierarchical manner. To some extent, this also allows safe failure of actors, adopting Erlang's "let it crash" philosophy.

Closer to our work is [29], in which Mostrous and Vasconcelos present a static session type verification mechanism for a small fragment of the Core Erlang language. Core Erlang is dynamically typed, akin to Elixir. In [29], each message sent or received is labelled with unique references. Then, using correlation sets, each message is matched with the corresponding session, given that multiple sessions are allowed to be running simultaneously. Its typing system is not algorithmic, so there is no implementation. We take a more pragmatic approach, adding more flexibility (*e.g.*, inductive types and infinite computation using recursion) and a larger part of the language (*e.g.*, variable binding with sequencing and expressions).

Other works have statically analysed Erlang for compile-time errors. Harrison [14] statically checks Core Erlang code for message passing errors. For example, his tool matches send statements with corresponding receive statements, thus ascertaining that orphan messages (*i.e.*, messages that are never received) can never exist. The tool works fully automated, so no annotations are required. In another implementation, Harrison [15] performs static and runtime analysis in Erlang. In [15], messages analysed were produced by OTP behaviours, *e.g.* `gen_server` or `gen_statem` behaviours, rather than send or receive statements directly. It works by intercepting messages dynamically and checks that each message has the expected type. Malformed messages are caught earlier, preventing any unexpected bahaviour.

Several works have been attempted to statically typecheck fragments of Erlang. Rajendrakumar and Bieniusa [37] created a bidirectional type system which uses both type inference and type checking to ensure that an Erlang program is well-typed. Compared to our tool, this type system [37], along with other Erlang type checkers [27, 42], tend to omit static concurrency checks. For example, they do not check whether the types of messages sent match the types of messages a process expects to receive.

Scala. Scalas and Yoshida [38] provide a library that checks binary session types in the Scala language. By abstracting session types as a set of Scala classes, the compiler is able to statically detect protocol errors. Linearity checks are performed at runtime. In another implementation, Scalas

[5]https://hexdocs.pm/ex_unit/ExUnit.html

et al. [39, 40] extended Dotty (Scala 3) by utilising dependent function types to verify programs at compile-time. Checking conformance to the expected behaviour is done via model checking. Bartolo Burlò et al. [5] synthesise runtime monitors from session types. They leverage the work of Scalas and Yoshida [38] to dynamically check binary sessions in Scala at runtime.

Other Actor Languages. Harvey et al. [16] present a new actor-based language, called EnsembleS, having native session type features. It is able to statically guarantee correctness while also allowing dynamic adaptation of actors. This allows actors to establish a connection and terminate mid-session, given that they obey a known protocol.

De'Liguoro and Padovani [10] used a different approach to type concurrent processes. Instead of relying directly on actors, they introduced the *mailbox calculus* which treats mailboxes as first-class citizens. Using mailbox types, which are a different kind of behavioural types, the state of mailboxes are typechecked to ensure deadlock freedom of processes.

5.2 Other Session Type Implementations

The remaining implementations are built solely on channel-based message passing, which contrast to our work, being actor-based.

Rust. Jespersen et al. [21] leverage Rust's affine type system and annotations (*e.g.*, `#[must_use]`) to implement binary session types. This implementation has some limitations, including that branches and choices are limited to binary options, and sessions can be unsafely dropped prematurely. Kokke's [22] work improves on [21] by basing their work on Exceptional GV [12], adding support for early cancellation of sessions. Lagaillardie et al. [24] extend [22] to implement a multiparty session type implementation for Rust. Another implementation for Rust was presented by Cutner and Yoshida [9], in which they use asynchronous communication rather than synchronous.

Haskell and OCaml. There are several session type implementations for Haskell, including [36] which uses monads and [23] which makes use of Linear Haskell and priorities guaranteeing compile-time linearity.

Padovani [34] presents FuSe, a simple library that performs static session typechecking for OCaml. It enforces linearity at runtime, similar to [38]. Melgratti and Padovani [28] extend FuSe to perform additional runtime checks, referred to as contracts. Contracts can be written natively in OCaml, so they can be inserted directly within the original program to form an inline monitor. Another approach uses parametric polymorphism, in which Imai et al. [20] propose a static binary session type system for OCaml. This was later expanded by the same authors in [19] by utilising global combinators to statically verify multiparty session types. The approaches for OCaml and Haskell rely heavily on type-level features of the language, which do not translate easily to Elixir.

5.3 Type System for Elixir

There have been other attempts to statically type the Elixir language, which is dynamically typed. In [6, 7], Cassola et al. created a gradual static type system for Elixir, which statically typechecks part of Elixir while permitting untyped parts. This work checks the functional part of the language, rather than the concurrent part. They analyse the AST directly from the Elixir source *without* macro expansions. In contrast, we statically analyse the AST *with* the macros fully expanded, as discussed in Sec. 4.1. This gives us several benefits, including having to explicitly check for less constructs, *e.g.*, analysing the **`case`** statement gave us other constructs for free as a side effect – the **`if`** and **`unless`** macros are directly expanded into a **`case`** construct. Also, this allows for more flexibility, since we are able to use custom macros, given that they will always be expanded during compilation. Macros in Elixir are useful as they can extend the Elixir language with first-class features, helping to create more concise programs [26].

6 Conclusion

In this paper we presented our preliminary work towards applying session types for Elixir. We focused on the adaptation of binary session types for the concurrent part of the Elixir language, making design decisions based on the language itself (*e.g.*, recursion in the form of function calls). This resulted in a tool implementation called ElixirST, which can statically typecheck public functions within a module against session-type usage specifications.

Future Work. We plan to extend our type system to be able to handle code *across* modules. A further natural extension of this work would then be the handling of multiparty session types. In order to achieve this, we can use Scribble [18] to generate local session types for individual functions and check them statically, similar to [16, 19, 24].

We also plan to prove a number of properties about our type system, which may require tweaking certain typing rules. We are in the process of formulating a reduction semantics for our targeted language subset that faithfully models the runtime behaviour of Elixir programs; this will in turn enable us to prove properties such as type preservation.

Acknowledgments

The work has been partly supported by the project MoVeMnt (No: 217987-051) under the Icelandic Research Fund, the project "Monitored Behavioural APIs for the API Economy" (No: CPSRP01-21) under the University of Malta Research Fund, and the BehAPI project funded by the EU H2020 RISE under the Marie Skłodowska-Curie action (No: 778233).

References

[1] Gul A. Agha. 1990. *ACTORS - a model of concurrent computation in distributed systems.* MIT Press.

[2] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Found. Trends Program. Lang.* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

[3] Joe Armstrong, Robert Virding, and Mike Williams. 1993. *Concurrent programming in ERLANG.* Prentice Hall.

[4] Duncan Paul Attard, Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. 2021. Better Late Than Never or: Verifying Asynchronous Components at Runtime. In *FORTE (Lecture Notes in Computer Science, Vol. 12719)*. Springer, 207–225.

[5] Christian Bartolo Burlò, Adrian Francalanza, and Alceste Scalas. 2021. On the Monitorability of Session Types, in Theory and Practice. In *ECOOP (LIPIcs, Vol. 194)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:30.

[6] Mauricio Cassola, Agustín Talagorria, Alberto Pardo, and Marcos Viera. 2020. A Gradual Type System for Elixir. *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity* (Oct 2020). https://doi.org/10.1145/3427081.3427084

[7] Mauricio Cassola, Agustín Talagorria, Alberto Pardo, and Marcos Viera. 2021. A Gradual Type System for Elixir. *CoRR* abs/2104.08366 (2021). arXiv:2104.08366 https://arxiv.org/abs/2104.08366

[8] David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 29:1–29:30.

[9] Zak Cutner and Nobuko Yoshida. 2021. Safe Session-Based Asynchronous Coordination in Rust. In *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12717)*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer, 80–89. https://doi.org/10.1007/978-3-030-78142-2_5

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

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

[12] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.* 3, POPL (2019), 28:1–28:29. https://doi.org/10.1145/3290341

[13] Qiang Guo, John Derrick, Clara Benac Earle, and Lars-Åke Fredlund. 2010. Model-Checking Erlang - A Comparison between EtomCRL2 and McErlang. In *TAIC PART (Lecture Notes in Computer Science, Vol. 6303)*. Springer, 23–38.

[14] Joseph Harrison. 2018. Automatic detection of core Erlang message passing errors. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang, ICFP 2018, St. Louis, MO, USA, September 23-29, 2018*, Natalia Chechina and Adrian Francalanza (Eds.). ACM, 37–48. https://doi.org/10.1145/3239332.3242765

[15] Joseph Harrison. 2019. Runtime type safety for erlang/otp behaviours. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang, Erlang@ICFP 2019, Berlin, Germany, August 18, 2019*, Adrian Francalanza and Viktória Fördős (Eds.). ACM, 36–47. https://doi.org/10.1145/3331542.3342571

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

[17] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 715)*, Eike Best (Ed.). Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[18] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. Scribbling Interactions with a Formal Foundation. In *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6536)*, Raja Natarajan and Adegboyega K. Ojo (Eds.). Springer, 55–75. https://doi.org/10.1007/978-3-642-19056-8_4

[19] Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. 2020. Multiparty Session Programming With Global Protocol Combinators. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.9

[20] Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2019. Session-ocaml: A session-based library with polarities and lenses. *Sci. Comput. Program.* 172 (2019), 135–159. https://doi.org/10.1016/j.scico.2018.08.005

[21] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, Patrick Bahr and Sebastian Erdweg (Eds.). ACM, 13–22. https://doi.org/10.1145/2808098.2808100

[22] Wen Kokke. 2019. Rusty Variation: Deadlock-free Sessions with Failure in Rust. In *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019 (EPTCS, Vol. 304)*, Massimo Bartoletti, Ludovic Henrio, Anastasia Mavridou, and Alceste Scalas (Eds.). 48–60. https://doi.org/10.4204/EPTCS.304.4

[23] Wen Kokke and Ornela Dardha. 2021. Deadlock-Free Session Types in Linear Haskell. *CoRR* abs/2103.14481 (2021). arXiv:2103.14481 https://arxiv.org/abs/2103.14481

[24] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2020. Implementing Multiparty Session Types in Rust. In *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12134)*, Simon Bliudze and Laura Bocchi (Eds.). Springer, 127–136. https://doi.org/10.1007/978-3-030-50029-0_8

[25] Tobias Lindahl and Konstantinos Sagonas. 2006. Practical type inference based on success typings. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, Annalisa Bossi and Michael J. Maher (Eds.). ACM, 167–178. https://doi.org/10.1145/1140335.1140356

[26] Chris MacCord. 2015. *Metaprogramming Elixir - Write Less Code, Get More Done (and Have Fun!).* O'Reilly. http://www.oreilly.de/catalog/9781680500417/index.html

[27] Simon Marlow and Philip Wadler. 1997. A Practical Subtyping System For Erlang. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman (Eds.). ACM, 136–149. https://doi.org/10.1145/258948.258962

[28] Hernán C. Melgratti and Luca Padovani. 2017. Chaperone contracts for higher-order sessions. *Proc. ACM Program. Lang.* 1, ICFP (2017), 35:1–35:29.

[29] Dimitris Mostrous and Vasco Thudichum Vasconcelos. 2011. Session Typing for a Featherweight Erlang. In *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6721)*, Wolfgang De Meuter and Gruia-Catalin Roman (Eds.). Springer, 95–109. https://doi.org/10.1007/978-3-642-21464-6_7

[30] Rumyana Neykova. 2013. Session Types Go Dynamic or How to Verify Your Python Conversations. In *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013 (EPTCS, Vol. 137)*, Nobuko Yoshida and Wim Vanderbauwhede (Eds.). 95–102. https://doi.org/10.4204/EPTCS.137.8

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

[32] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. 2013. SPY: Local Verification of Global Protocols. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8174)*, Axel Legay and Saddek Bensalem (Eds.). Springer, 358–363. https://doi.org/10.1007/978-3-642-40787-1_25

[33] Sven-Olof Nyström. 2003. A soft-typing system for Erlang. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang, Uppsala, Sweden, August 29, 2003*, Bjarne Däcker and Thomas Arts (Eds.). ACM, 56–71. https://doi.org/10.1145/940880.940888

[34] Luca Padovani. 2017. A simple library implementation of binary sessions. *J. Funct. Program.* 27 (2017), e4.

[35] Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

[36] Riccardo Pucella and Jesse A. Tov. 2008. Haskell session types with (almost) no class. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, Andy Gill (Ed.). ACM, 25–36. https://doi.org/10.1145/1411286.1411290

[37] Nithin Vadukkumchery Rajendrakumar and Annette Bieniusa. 2021. Bidirectional typing for Erlang. In *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang, Erlang@ICFP 2021, Virtual Event, Korea, August 26, 2021*, Stavros Aronis and Annette Bieniusa (Eds.). ACM, 54–63. https://doi.org/10.1145/3471871.3472966

[38] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:28. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

[39] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Effpi: verified message-passing programs in Dotty. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019*, Jonathan Immanuel Brachthäuser, Sukyoung Ryu, and Nathaniel Nystrom (Eds.). ACM, 27–31. https://doi.org/10.1145/3337932.3338812

[40] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying message-passing programs with dependent behavioural types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 502–516. https://doi.org/10.1145/3314221.3322484

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

[42] Nachiappan Valliappan and John Hughes. 2018. Typing the wild in Erlang. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang, ICFP 2018, St. Louis, MO, USA, September 23-29, 2018*, Natalia Chechina and Adrian Francalanza (Eds.). ACM, 49–60. https://doi.org/10.1145/3239332.3242766

[43] Vasco T. Vasconcelos. 2012. Fundamentals of session types. *Inf. Comput.* 217 (2012), 52–70. https://doi.org/10.1016/j.ic.2012.05.002